UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA EN TELECOMUNICACIONES

PROYECTO FIN DE CARRERA

DESARROLLO Y VALIDACIÓN DE UN BOOTLOADER PARA APLICACIONES DE CARGA REMOTA EN ENTORNOS INALÁMBRICOS 802.15.4 Y ZIGBEE

AUTOR: Jorge Jiménez Würzburger

TUTOR: Dr. José Ignacio Moreno Novella

19 Noviembre 2009

PROYECTO FIN DE CARRERA

Departamento de Ingeniería Telemática

Universidad Carlos III de Madrid

Título: Desarrollo y validación de un bootloader para aplicaciones de carga remota en entornos inalámbricos IEEE 802.15.4 y ZigBee.

Autor: Jorge Jiménez Würzburger.

Tutor: Dr. José Ignacio Moreno Novella.

EL TRIBUNAL

Presidente: D. Mario Muñoz Organero

Secretario: D. Ángel Cuevas Rumín

Vocal: Dña. Matilde Sánchez Fernández

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 19 de Noviembre de 2009 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

Presidente Secretario Vocal

Resumen

En el entorno hostil que suele representar una red inalámbrica de sensores (WSN): escenarios con alta densidad de dispositivos, equipos instalados en localizaciones de difícil acceso, comunicaciones de alta latencia sobre canales ruidosos con numerosas interferencias, etcétera, se hace necesario un mecanismo que posibilite la actualización del *firmware* de los componentes del sistema. De esta forma se podrían corregir errores de funcionamiento, modificar parámetros de comportamiento o simplemente añadir nuevas funcionalidades a los dispositivos de la red.

La tecnología ZigBee, diseñada para la intercomunicación inalámbrica de dispositivos de reducida capacidad, tamaño, consumo y coste, se plantea como una de las más apropiadas para cubrir la mayoría de las necesidades demandadas en el campo de las redes de sensores.

Sin embargo ni el estándar ZigBee, ni los perfiles superiores de aplicación que funcionan sobre él, especifican el procedimiento que han de seguir los dispositivos para actualizar su *firmware*, dejando que cada fabricante plantee su solución propietaria para los productos que introduce en el mercado.

Así pues, el objetivo de este proyecto será desarrollar un cargador de arranque (**bootloader**) capaz de sustituir el *firmware* en ejecución en dispositivos ZigBee por otro que previamente una aplicación de carga remota ha recibido y almacenado en una memoria auxiliar.

Para ello, será necesario familiarizarse previamente con el entorno de desarrollo (*CodeWarrior*) y las librerías IEEE 802.15.4 y ZigBee que suministra uno de los principales fabricantes de chips en la banda de frecuencias de 2,4GHz, *Freescale*, para una vez adquirida la capacidad de crear nuevas aplicaciones sobre ellas y entender el funcionamiento de las diferentes capas, programar el bootloader.

Puesto que el procedimiento interno de actualización del *firmware* de un dispositivo ZigBee es muy dependiente de los componentes (chip, memoria, bus, etc.) que integre, será necesario estudiar también, a lo largo del desarrollo del proyecto, el caso concreto de los circuitos de los equipos que pretenden soportar esta aplicación.

Finalmente, puesto que el bootloader debe ser capaz de actualizar equipos ZigBee de forma independiente del modo en que la nueva imagen llegue hasta la memoria auxiliar, será imprescindible elaborar una aplicación auxiliar (**cargador RS232**) capaz de transmitir el nuevo *firmware* al dispositivo. La elaboración de este cargador permitirá depurar y validar el funcionamiento del bootloader además de evaluar el formato óptimo de almacenamiento de la nueva imagen de *firmware* en la memoria auxiliar.

Abstract

In the hostile environment that is often a wireless sensor network (WSN), a great need of a system devices' *firmware* updating mechanism rises up. This way, application *bugs* could be fixed, devices' behaviour could be modified and new functionalities could be added.

ZigBee technology is designed to grant wireless communications between small, low consumption & low cost devices, so it is probably the most appropriate standard to meet the needs demanded in the field of sensor networks.

However, neither the ZigBee standard nor the application profiles specify how wireless devices can update their *firmware*. So each manufacturer will have to develop its proprietary solution for the devices that sells.

Thus, the objective of this final thesis is to develop a **boot loader** capable of replacing the old *firmware* running on a ZigBee device by a new one previously stored in an auxiliary EEPROM by a remote programming application.

To achieve this aim, the first step to be taken will be choosing a suitable ZigBee chip manufacturer (*Freescale*) and then becoming familiar with its development environment (*CodeWarrior*) and its IEEE 802.15.4/ZigBee libraries.

Freescale is nowadays one of the leading ZigBee chip manufacturers working on the frequency band of 2.4 GHz and, at the time this thesis started, it was probably the best choice.

Since the internal process of updating the *firmware* of a ZigBee device is highly dependent on its integrated components (chip, memory, bus, etc...) a deep study of chosen ZigBee device's integrated circuits will have to be done next in order to successfully develop the bootloader.

Finally, since the developed application should be able to update ZigBee devices in real time independently of the means the new image is stored in the EEPROM. To achieve that, it will be essential to develop next an auxiliary application (**RS232 loader**) capable of transmitting the new *firmware* to the device so the programmed bootloader could be debugged and validated.

In addition to all this, the optimum format for storing the new *firmware* image in the auxiliary memory will be discussed.

Agradecimientos

Cuando estoy por fin a punto de acabar esta etapa de mi vida, que en ocasiones pensé que nunca lograría terminar, no me queda ya más que agradecer a todos aquellos que han hecho que este momento sea posible.

En primer lugar quiero agradecer a mis padres y a mi hermana su apoyo y comprensión incondicional en todo momento, incluso en las ocasiones en que había perdido completamente el rumbo. Si estoy ahora aquí es gracias a vosotros.

Igualmente gran parte del mérito es de Sara, mi paciente compañera de los últimos años, cuyos fuertes puntapiés en mi trasero han sido sin ninguna duda el acicate que necesitaba para dar el último paso de terminar esta memoria. Muchísimas gracias por soportarme y apoyarme a lo largo todo el proceso. Ha sido un privilegio compartirlo contigo.

Un agradecimiento especial merece también toda la gente maravillosa que he tenido la oportunidad de conocer en la universidad. Mis amigos de los primeros años: Carlos, Edu, Dani, Raúl, Emilio, Nacho y Sara, Chico, Elisa, Cano, Iván, Álvaro y Raquel, Guillermo y Sara, Gabi, Noelia, Adrián, Pablo, Patri, Ruth, Sergi, Turrin, Sonia, Natalia, Eva, Sergio y Ramón, con los que he superado muchos malos momentos y disfrutado aún muchos más buenos.

Mis amigos de los últimos años, responsables en gran medida de que volviese a disfrutar con la carrera en la mejor compañía. Nunca se me van a olvidar todas esas situaciones de risas histéricas con vosotros en la biblioteca, a la una de la mañana, la víspera de los exámenes. Muchas gracias Aitor, Marcelo, Arry, Fito, Miri, Merche, Mele, Iván, Ivanga, Jauma, Ysus, Inma, Joseto, Davo, RAM, Sofi, Isidro, Marta, Nere, David y Demelza.

Mis compañeros de la asociación, con los que he compartido infinidad de momentos buenos, merecen también una mención especial. Gracias Pablo (Presi), Raquel, Juanvi, Carlitos, Cortés, Carlos, Isaac, Álvaro, Chus, Sacha, Álex, Gorka y Morán.

Mis compañeros del servicio informático también aportaron su granito de arena a todo este proyecto, de modo que mil gracias Pay, Movi, Iván, David, Mayte, Chechu, Amarri y Jeda.

A mis amigos del barrio, que pacientemente han soportado mis continuos desplantes a lo largo de años sin mandarme al cuerno. Ahora que me quedo sin excusas para seguir siendo un mal amigo, voy a tener que hacer algo al respecto. Muchas gracias Pablo, Irene, Jose, Gonzalo, Miguel, Darío, Rubén, Raquel, Bedoya, Fiz y Peni.

Mis compañeros de NLaza, junto a los que he aprendido mucho, merecen también mi agradecimiento. Gracias muchas pues, señores David (qué grandes nuestras conversaciones), Pedro y José.

También quiero agradecer a mi tutor, José Ignacio Moreno, que me ha ayudado a terminar este proyecto a pesar de la tremenda informalidad a la que le he sometido. En serio, muchísimas gracias.

Igualmente quiero agradecerle a su equipo, Grego, Gema, y Víctor, todas las risas compartidas y el apoyo mostrado.

En último lugar, muchas gracias también a todos aquellos que no creísteis que lo conseguiría, fuisteis la inspiración que necesitaba.

Índice

Capítulo 1
Motivación y Objetivos

1.1. Motivación

Las redes de sensores llevan siendo objeto de investigación y estudio desde hace ya más de una década. Las relativamente recientes publicaciones de estándares y soluciones en este campo, como pueden ser IEEE 802.15.4 [1], ZigBee [5], Z-Wave [7], Bluetooth ULP [8], etc., están propiciando la salida de estas tecnologías de los laboratorios de investigación hacia el mercado de consumo.

Aunque las primeras investigaciones sobre redes de sensores estaban orientadas principalmente hacia aplicaciones militares, hoy en día se conciben aplicaciones mucho más diversas como son la automatización del hogar y edificios, la automatización industrial o la lectura automática de contadores, por ejemplo.

Debido entre otras cosas a los retrasos producidos en la publicación de especificaciones y estándares de protocolos de comunicación para redes de sensores, ha tenido lugar la aparición de diversas tecnologías propietarias que pretenden extender su solución particular en el mercado.

El estándar IEEE 802.15.4 para comunicaciones inalámbricas a 250 Kbps en la banda sin licencia de 2.4GHz (y sus alternativas de menor velocidad de transferencia en las bandas de 868 y 900 MHz), así como la posterior especificación ZigBee, han supuesto unas de las más robustas alternativas de acceso público ante la amplia oferta de tecnologías propietarias.

Esta característica (estándar de público acceso) convierte a ZigBee en una opción muy atractiva para los desarrolladores de productos inalámbricos, ya que asegurará la interoperabilidad de sus dispositivos con los de otras marcas. Por otro lado también será atractiva para los clientes finales, ya que no dependerán de una única compañía y podrán disponer de diferentes opciones a la hora de ampliar su red.

El estándar IEEE 802.15.4 es un protocolo abierto diseñado para posibilitar una comunicación fiable y segura entre dispositivos de bajo consumo (muchas veces a pilas y dormidos la mayor parte del tiempo) que forman redes en forma de árbol, estrella o malla y trabajan sobre bandas de frecuencia gratuitas pero colmadas de interferencias. Para poder garantizar dicha fiabilidad, el estándar provee de un protocolo muy robusto frente a las perturbaciones producidas por otras comunicaciones o emisiones de dispositivos domésticos basado en técnicas de espectro ensanchado (DSSS), control de acceso al medio y detección de colisiones.

ZigBee se presenta como una especificación no propietaria de las capas de red, transporte y aplicación que habrá de funcionar sobre las capas física y de acceso al medio (MAC) que ofrece el estándar IEEE 802.15.4. De este modo, se beneficia de sus propiedades y aporta

valor añadido al proporcionar soluciones ínter operables, de bajo coste y bajo consumo en entornos de automatización del hogar, de edificios, industrial, de lectura automática de contadores (AMR) y un largo etcétera.

La *ZigBee Alliance* [9] es una asociación de más de 225 compañías de 28 países distintos que trabajan de forma conjunta para posibilitar la elaboración de productos fiables, económicos y de bajo consumo que, interconectados mediante red inalámbrica, permitan monitorización y control y se basen todos ellos en el mismo estándar global abierto.

Los miembros actuales de la *ZigBee Alliance* son proveedores y fabricantes de tecnología en el todo el mundo lo cual, junto con la extensa gama de transceptores en las bandas de frecuencia de 800, 900 y 2400MHz disponibles en el mercado actual, posibilita la aparición inminente de una gran variedad de productos ZigBee robustos, relativamente baratos y fiables.

Con todo esto el estándar ZigBee parece ser, a día de hoy, una de las tecnologías más interesantes para el desarrollo de aplicaciones que funcionen sobre redes de sensores inalámbricas, formadas por dispositivos de bajo consumo y coste reducido. Esto se debe a que:

- Trabaja sobre el estándar público IEEE 802.15.4, diseñado por una de las entidades certificadoras más importantes y sólidas del mundo.
- La *ZigBee Alliance*, responsable de la elaboración de la especificación ZigBee, es una asociación de compañías muy representativas del sector (tales como *Philips*, *Texas Instruments, Motorola, Siemens* o *Honeywell*) siempre interesadas en sacar un producto funcional y sólido al mercado que será continuamente perfeccionado para cubrir todas las necesidades que vayan surgiendo. A día de hoy continúa publicando revisiones y mejoras de la especificación, así como diferentes perfiles de aplicaciones (*Home Automation* [10], *Smart Energy* [11], *Building Automation, Automatic Meter Reading*) para cubrir las necesidades emergentes del mercado de redes de sensores [12] [13].
- Al ser una especificación pública, muchas compañías suministran kits de desarrollo que incluyen la pila de protocolos IEEE 802.15.4 y ZigBee que permite implementar aplicaciones sobre sus chips de la forma más rápida posible [14] [15] [16]. De esta forma, es relativamente sencillo empezar a desarrollar aplicaciones ZigBee una vez se ha elegido el chip y el transceptor radio de un fabricante determinado.

Una vez elegida la tecnología ZigBee como protocolo de comunicaciones inalámbricas en un entorno de redes de sensores de bajo consumo y coste reducido, es preciso preguntarse qué tipo de aplicaciones funcionarán sobre ésta. Para ello la especificación ZigBee soporta una serie de perfiles de aplicación (*profiles*) que definen el formato de un conjunto de comandos,

atributos y variables que serán necesarios para que una aplicación determinada funcione correctamente sobre una red de sensores formada por dispositivos de diferentes fabricantes.

Estos perfiles son desarrollados y mantenidos por las compañías integrantes de la *ZigBee Alliance* y responden a las necesidades de una red de sensores de bajo consumo soportando muy diversas aplicaciones destinadas a entornos como podrían ser la automatización del hogar, la automatización industrial o la lectura automática de contadores.

De este modo, el perfil del *Home Automation* [10] (primer perfil ZigBee publicado por la *ZigBee Alliance* en el año 2007 y destinado a la automatización del hogar [17]), incluye las especificaciones necesarias que tendrán que cumplir dos (o más) dispositivos de diferentes fabricantes para poder funcionar de forma conjunta. Ejemplos serían pues un interruptor inalámbrico y una fuente de luz regulable, un sensor de temperatura y un termostato, etc.

En este escenario de aplicaciones y dispositivos tan diversos, se hace necesario un mecanismo que permita poder actualizar el *firmware* de los equipos, ya sea para reemplazar el código que se está ejecutando (por ejemplo en el caso de la automatización del hogar, actualizar un interruptor OnOff a un interruptor regulable), para añadir nuevas funcionalidades, o simplemente para corregir errores de la aplicación.

Hay que tener en cuenta que las redes de sensores inalámbricas pueden presentar una alta densidad de dispositivos (lo cual puede complicar su identificación) y estar posicionados en localizaciones de difícil acceso como pueden ser techos, cajas de registro o tomas eléctricas. Idealmente, en una red ZigBee, cada sensor dispondrá del mínimo número indispensable de periféricos y puntos de acceso integrados ya que eso reducirá su coste y consumo al máximo.

Todos estos factores, junto con el hecho de que de momento no existe un método de actualización estandarizado en la especificación ZigBee o en los perfiles de aplicación (*profiles*), propiciarán que cada fabricante diseñe su método propietario para actualizar el *firmware* de los dispositivos que suministra.

De todas las soluciones posibles aplicables a un entorno tan heterogéneo como puede llegar a ser una red ZigBee, la solución inalámbrica se plantea claramente como la opción más interesante ya que cubre el mayor número de situaciones posibles. Esto se debe a que todos los dispositivos dispondrán obligatoriamente de una antena y serán capaces de comunicarse mediante el mismo protocolo dentro de las mismas bandas de frecuencia.

El proceso de actualización inalámbrica (*Over the Air Programming* [18]) del *firmware*, se puede separar siempre en dos partes perfectamente diferenciables:

1. **Carga Remota**: protocolo de transferencia del nuevo *firmware* desde un dispositivo emisor a uno o varios receptores a través de la red para su posterior almacenamiento. Este protocolo será siempre independiente del hardware de los dispositivos receptores y podría permitir a dos fabricantes distintos actualizar sus equipos de forma idéntica siempre que implementen un algoritmo común en sus aplicaciones.
2. **Bootloader**: aplicación responsable de la actualización del *firmware* antiguo por el nuevo en tiempo de ejecución y de la reinicialización del dispositivo. Este procedimiento es fuertemente dependiente del hardware implicado y será específico de cada modelo de dispositivo ZigBee.

Este proyecto pretende desarrollar pues un bootloader, que será capaz de sustituir una aplicación en ejecución en un dispositivo ZigBee por otra nueva almacenada previamente en su memoria auxiliar. El protocolo elegido para que dicha actualización sea transmitida y almacenada en el dispositivo será trasparente al funcionamiento del bootloader (pero el conjunto deberá ser compatible).

1.2. Objetivos

Este proyecto surge como una colaboración entre la Universidad Carlos III de Madrid y la empresa *NLaza Soluciones* [19] para la elaboración conjunta de una solución capaz de actualizar el *firmware* de los dispositivos 802.15.4 y ZigBee suministrados por esta última.

De esta forma, el desarrollo de esta solución se dividirá en dos secciones: la relativa a la transferencia inalámbrica e interfaz con el usuario (carga remota) y la relativa al almacenamiento, actualización del *firmware* y reinicialización del dispositivo (bootloader). Ambas partes son perfectamente diferenciables y es esta última, el bootloader, el objetivo final de este proyecto.

Para lograr dicho objetivo será necesario estudiar en primer lugar el funcionamiento del estándar IEEE 802.15.4 y de la especificación ZigBee. De este modo se podrá después entender adecuadamente la implementación que de estos protocolos hacen las librerías de *Freescale* (compañía que formaba parte de *Motorola* hasta el año 2004 y que es uno de los principales fabricantes de chips en la banda de frecuencia de 2.4GHz; chips que además están integrados en las placas de NLaza Soluciones).

Posteriormente seguirá un proceso de familiarización con el entorno de desarrollo y las librerías de *Freescale* tras el cual, una vez adquirida la capacidad de crear nuevas aplicaciones y entender la comunicación existente entre las diferentes capas, se procederá a desarrollar un bootloader que posibilite la actualización del *firmware* a través de distintos mecanismos: puertos serie, interfaz radio, etc.

El bootloader desarrollado deberá ser capaz de desempeñar las siguientes funciones ante una solicitud emitida desde la aplicación anfitrión: acceder a la imagen del nuevo *firmware* almacenado previamente en una memoria EEPROM (integrada en los dispositivos NLaza Soluciones y accesible mediante el protocolo I^2C), interpretar su formato, borrar la memoria Flash, sobrescribirla con la imagen almacenada, realizar las comprobaciones de seguridad y validación pertinentes y, en último lugar, reinicializar el sistema.

Puesto que tanto las partes relativas a la carga remota como la relativa al bootloader (este proyecto) serán desarrolladas de forma simultánea e independiente por dos equipos de trabajo diferentes, el siguiente paso a dar será la elaboración de un cargador serie (RS232) que se encargue de preparar imágenes de prueba en la memoria auxiliar del dispositivo.

Esta aplicación auxiliar permitirá ir depurando la aplicación desarrollada así como ir evaluando y optimizando el formato de almacenamiento de las imágenes en el equipo receptor.

Cerca del final del trabajo se analizará el hardware integrado en los equipos ZigBee suministrados por NLaza Soluciones para poder configurar los puertos y periféricos de forma que el bootloader pueda: acceder al nuevo *firmware* preparado en la memoria auxiliar, regrabar la memoria del dispositivo, reinicializar la placa y permitir al resto de las aplicaciones trabajar de forma transparente mientras no se está actualizando el dispositivo.

Así pues el objetivo final de este proyecto será obtener un bootloader capaz de actualizar el *firmware* de un dispositivo ZigBee de forma fiable, robusta, transparente al resto de aplicaciones y que además sea independiente del mecanismo (inalámbrico o no) que se emplee para hacer llegar la nueva imagen al dispositivo receptor.

La única condición necesaria para esta transmisión será que la nueva imagen termine alojándose en la memoria EEPROM auxiliar siguiendo un formato concreto que se acordará de antemano. Dicho formato se diseñará de forma que optimice el espacio disponible en la EEPROM dadas las restricciones existentes de memoria, batería y capacidad de proceso.

1.3. Organización de la Memoria

Para entender correctamente la estructura de este proyecto, es necesario tener en cuenta que se ha dedicado tanto tiempo al estudio "teórico" de las redes ZigBee y a los módulos contenidos en el chip elegido, como al trabajo "práctico" que ha supuesto la familiarización con el entorno de desarrollo, la programación y depuración de las aplicaciones, y la validación de éstas.

Así pues, la organización de la memoria se ha llevado a cabo en varios capítulos que aparecen ordenados de forma más o menos cronológica en el índice de este documento.

De este modo, los primeros capítulos serán principalmente teóricos, ya que describirán el estado del arte de la tecnología y las características de la plataforma finalmente seleccionada (y las razones que justifican dicha elección).

Los siguientes capítulos se centrarán en las herramientas y protocolos con los que habrá que familiarizarse para desarrollar la aplicación objetivo de este proyecto (capítulo 3, escenario de aplicación) y en el estudio de los módulos de la plataforma MC13213 directa o indirectamente relacionados con dicha aplicación (primeros subapartados del capítulo 4).

Inmediatamente después, se explicará el diseño, desarrollo y la validación de las aplicaciones del bootloader y del cargador RS232 (resto del capítulo 4 y el capítulo 5). Este bloque representará el grueso del trabajo "práctico" realizado.

Finalmente se desarrollarán las conclusiones obtenidas fruto del desarrollo de este proyecto y se esbozarán los posibles trabajos futuros que podrían surgir a raíz de éstas (capítulo 6 y apéndices).

A continuación se ofrece un breve resumen de cada capítulo para facilitar una visión global del trabajo realizado.

Capítulo 1: Motivación y Objetivos

En este capítulo se introducen las necesidades relacionadas con la carga remota a las que es necesario dar solución y se concretan exactamente los objetivos que se pretenden alcanzar con este proyecto a ese respecto.

Adicionalmente, se explicará la distribución de esta memoria y se presentarán los medios materiales que han sido necesarios para el desarrollo efectivo del proyecto.

Capítulo 2: Estado del Arte

En el capítulo 2 se introducirán de forma resumida las tecnologías inalámbricas de bajo consumo de redes de sensores disponibles en la actualidad (y en el momento concreto en que se inició este proyecto) para luego estudiar de forma más profunda los estándares 802.15.4 y ZigBee, sobre los que se basará este proyecto.

A continuación se procederá a realizar una evaluación de las plataformas ZigBee disponibles en el mercado a principios del 2007 (y en la actualidad) y a justificar la elección de una de ellas: *Freescale*.

Finalmente se introducirán algunas de las soluciones existentes para la autoconfiguración de dispositivos de comunicaciones. Se obtendrá así una idea general de lo que se trata de conseguir y los métodos que se emplean habitualmente para ello.

Capítulo 3: Escenario de Aplicación y Requisitos

Este capítulo se inicia con la exposición de los requisitos que una red de sensores de bajo consumo y que una aplicación de actualización de *firmware* han de satisfacer, ya que éstos serán luego objetivos primordiales del proyecto.

De estos requisitos se concluirá la necesidad de disponer de una aplicación abierta capaz de actualizar el *firmware* de los dispositivos en tiempo de ejecución. Sin embargo una parte fundamental de esta aplicación, el bootloader, es completamente dependiente del hardware elegido, de forma que será necesario seleccionar un producto comercial real para su desarrollo.

El dispositivo elegido finalmente será el ND07 de la línea de productos de NLaza Soluciones, que es un coordinador ZigBee con puerto RS232 para comunicaciones serie basado en el chip MC13213 de *Freescale*.

A continuación se analizarán el chip elegido (MC13213) y el dispositivo que lo integra (ND07), para tener consciencia de los módulos que los componen y que habrá que aprender a controlar.

Posteriormente se estudiará la librería ZigBee y el entorno de desarrollo ofrecidos por *Freescale* para adquirir la capacidad de programar nuevas aplicaciones ZigBee mediante ellas.

En último lugar se explicarán dos protocolos de comunicación, el bus I^2C y el protocolo RS232, que será imprescindible familiarizarse para implementarlos dentro de la aplicación final.

Capítulo 4: Bootloader y Aplicaciones Auxiliares

El capítulo 4 comenzará con una introducción al concepto de bootloader haciendo especial énfasis en la nomenclatura que se empleará en el resto de la memoria.

A continuación se desarrollará un estudio en profundidad de los módulos de la plataforma MC13213 de *Freescale* que están relacionados con el proyecto, y se explicarán las decisiones de diseño concluidas.

Más tarde se presentará la discusión mantenida respecto al formato óptimo (dadas las especificaciones de funcionamiento y las restricciones materiales) para el almacenamiento de las imágenes de *firmware* en la memoria auxiliar del ND07 y las conclusiones acordadas.

Finalizados todos los análisis se procederá explicar la estructura de la aplicación del bootloader, desarrollando cada uno de los módulos que la componen, el proceso de validación aplicado y las decisiones de diseño tomadas.

Dado que el bootloader se programó originalmente en C, y que más tarde se comprobó que para garantizar siempre su correcto funcionamiento era necesario reprogramarlo todo en ensamblador, los siguientes subapartados desarrollarán los elementos implicados (opciones del compilador, ensamblador, etc.) y el procedimiento seguido para realizar dicha “traducción”.

Las decisiones tomadas acerca del emplazamiento óptimo del bootloader en memoria (RAM y Flash) y dentro de la secuencia de arranque cerrarán la sección de la memoria relativa al diseño de esta aplicación, objetivo principal del proyecto.

Puesto que el bootloader es independiente del modo de transmisión del nuevo *firmware* hasta el dispositivo a actualizar, el resto del capítulo se dedicará a explicar el desarrollo de una aplicación auxiliar (Cargador Serie RS232), que hará uso de él con el fin de evaluar y optimizar su funcionamiento. Esta aplicación permitirá pues actualizar el *firmware* de dispositivos mediante imágenes enviadas a través de una comunicación serie RS232, empleando el bootloader para ello.

Capítulo 5: Validación del Sistema

En el capítulo 5 se presentarán en primer lugar los tres métodos empleados para depurar y validar tanto la aplicación del bootloader como la aplicación del cargador RS232. Estos métodos se aplicaron a lo largo de todo el proceso de desarrollo.

En último lugar se presentará el escenario propuesto para la validación del sistema completo, junto con la descripción del funcionamiento de dicho sistema obtenido en laboratorio.

Capítulo 6: Conclusiones y Trabajos Futuros

En el capítulo 6 se analizarán las conclusiones obtenidas fruto del desarrollo del proyecto y se plantearán las posibles líneas de trabajo futuro que podrían surgir a partir de los resultados alcanzados.

Apéndices

Los apéndices incluirán las instrucciones y recomendaciones de instalación del bootloader (una de las principales conclusiones de la memoria), el presupuesto requerido para realizar el proyecto y la bibliografía y referencias empleadas.

1.4. Medios Materiales para la Ejecución del Proyecto

La elaboración de este proyecto ha requerido de una serie de elementos hardware y software imprescindibles para alcanzar los objetivos impuestos. La justificación de la elección de la mayor parte de estos elementos (tecnología, hardware y empresas implicadas) vendrá dada en posteriores capítulos.

Podemos distinguir pues los siguientes componentes físicos (hardware):

- PC (Pentium 1GHz) con 512 MB RAM y 600MB de espacio en disco duro y puerto serie
- Kit de Desarrollo de *Freescale* para el chip MC1321x
- Dispositivo ZigBee con puerto serie RS232: NLaza NDimension ND07
- Adaptador de grabación para que el *USB MultiLink* de *Freescale* (incluido dentro del Kit de desarrollo) sea compatible con el ND07 (programador/depurador *NLaza-Freescale*)
- Dispositivo *sniffer* para la captura de tramas ZigBee: *Daintree Sensor Network Adapter*
- Fuente de alimentación regulable (para alimentar el ND07 durante los procesos de depuración y programación del dispositivo)

Las especificaciones del listado de dispositivos anterior las han dictado fundamentalmente los requisitos mínimos necesarios para el correcto funcionamiento de las aplicaciones implicadas en el desarrollo del bootloader.

Así pues enumeramos las aplicaciones (software) que han sido necesarias:

- Sistema Operativo Microsoft Windows® XP
- Entorno de Desarrollo (IDE): *CodeWarrior for Microcontrollers* (RS08/HC(S)08/ColdFire V1) con una licencia estándar
- Servidor de licencias flotantes *FlexLm* v8.4. (incluido con el CodeWarrior)
- Software Analizador de Protocolos (captura de tramas radio) *Daintree Sensor Network Analyzer* (SNA) con una licencia estándar
- Software *BeeKit Wireless Connectivity Toolkit* con una licencia estándar
- Pila 802.15.4 y ZigBee: *BeeStack 1.0.5*. (incluidos en el *BeeKit*)
- *WinMerge* 2.0.2 [20] Software libre para comparación de ficheros en Windows XP
- *RealTerm* 2.0.0.57 [21] Terminal serie (software libre) más potente que el *HyperTerminal* de Windows XP (versión por defecto)

Una vez elegida la tecnología inalámbrica (ZigBee), el fabricante de chips (*Freescale*) y el microprocesador/microcontrolador sobre el que se programará la aplicación del bootloader (plataforma MC13213), la mayor parte del software necesario para el desarrollo queda inevitablemente definido. Así pues, alrededor de la fechas en que se comenzó este estudio

(finales del 2006), *Freescale* suministraba el IDE *CodeWarrior for Microcontrollers (que incluía* el compilador necesario para el chip MC13213) junto con el servidor de licencias flotantes *FlexLm* dentro de su kit de desarrollo.

Contemporáneamente a la compra del kit de desarrollo del MC13213, se adquirió el analizador de redes (SNA) de la compañía *Daintree Network*. Esto se debió a que *Freescale* recomendaba sus herramientas, asegurando compatibilidad con sus chips, y existían pocas o ninguna opción adicional.

El *BeeKit* suministrado por *Freescale* es una herramienta diseñada para el desarrollo rápido y configuración de aplicaciones de ejemplo basadas en las pilas 802.15.4 y ZigBee, que han sido necesarias para la programación del bootloader y el cargador RS232. Así pues fue necesaria su adquisición también.

La elección de la empresa NLaza Soluciones como suministradora del hardware definitivo fue debida a que en 2006 se planteaba como una de las pocas empresas españolas desarrolladoras de dispositivos ZigBee y por entonces la única que trabajaba con *Freescale*. La elección del ND07 de entre su línea de productos fue debido a que era un dispositivo sencillo, con conexión serie RS232, y disponía de una memoria auxiliar EEPROM que permitirían el fácil desarrollo y testeo del bootloader (y de las aplicaciones auxiliares).

Finalmente, la elección de Windows como sistema operativo fue debida principalmente a la sencillez que ofrecía la configuración de las aplicaciones funcionando sobre Windows frente a la complejidad que presentaban las mismas sobre otras plataformas como Linux (sobre las que también funcionaban las aplicaciones *FlexLm* y *CodeWarrior*). Adicionalmente, por entonces existía más documentación y más herramientas para la depuración de código sobre Windows.

Capítulo 2
Estado del Arte

2.1. Introducción a las Redes de Sensores Inalámbricas

Los recientes avances en sistemas microelectromecánicos, miniprocesadores y tecnologías de transmisión inalámbrica de baja potencia en han permitido el desarrollo de una gran variedad de sensores de bajo coste, bajo consumo y reducido tamaño que son capaces de observar y reaccionar ante cambios de determinadas variables del entorno.

Cuando estos dispositivos se empleen de forma conjunta, colaborando y comunicándose de forma inalámbrica, se podrán obtener resultados con un mayor valor añadido, formando lo que se conoce como Redes de Sensores Inalámbricas (WSN).

Los sensores inalámbricos están equipados con un transceptor radio y una serie de transductores a través de los cuales obtienen información sobre el entorno que los rodea.

Cuando se despliegan en gran cantidad dentro de un escenario determinado, se organizarán automáticamente y configurarán una red *ad hoc* multisalto que les permitirá comunicarse entre ellos y con uno o más nodos sumidero (*sink*s). Estos sumideros permitirán a un usuario remoto la recolección de las medidas recibidas de los sensores de la red y la introducción de comandos destinados a éstos.

La aplicación auxiliar del cargador RS232 desarrollada en este proyecto (se explicará más adelante en la memoria), actuará como uno de estos nodos sumideros funcionando dentro una red de sensores ZigBee.

Las aplicaciones de esta tecnología son numerosas y pertenecen a campos muy diversos como pueden ser la medicina (monitorización de constantes vitales y envío de alarmas en caso de problemas), la agricultura (medición de condiciones climatológicas y control de cultivos), el medioambiente (medición de polución, detección y prevención de incendios e inundaciones, etc.), el control de inventarios, la detección de intrusos, la localización y seguimiento de personas, las aplicaciones militares, el control y monitorización de estructuras o maquinaria (puentes, aviones, etc.), juguetes y periféricos de PC, y un larguísimo etcétera [22] [23].

Existen dos campos de aplicación especialmente relevantes como son la automatización del hogar (domótica) y la gestión y medición remotas del consumo energético. Esto se debe principalmente a las optimistas estimaciones realizadas acerca de la relevancia que tendrán en el futuro las redes de sensores inalámbricas en dichos sectores. Para ambos campos de aplicación existe, en la actualidad, una considerable variedad de sensores disponibles en el mercado.

Sin embargo, los escenarios de aplicación de las redes de sensores suelen ser considerablemente complejos, debido principalmente a los problemas que arroja la coexistencia una alta densidad de dispositivos en un mismo área reducida, la alta latencia de las comunicaciones (los dispositivos entrarán en letargo habitualmente para ahorrar baterías), el reducido ancho de banda y las interferencias con otras tecnologías.

A estos inconvenientes habrá que sumarle las limitaciones impuestas por los dispositivos que integrarán habitualmente este tipo de redes, que presentarán un coste, tamaño y consumo reducidos, pero también una baja capacidad de cómputo y almacenamiento (memoria).

Es debido a todo esto que a lo largo de los últimos años han ido surgiendo tecnologías de redes de sensores muy diferentes y se hace cada vez más necesario el proceso de estandarización de los diversos protocolos de comunicación.

Así pues, se pueden distinguir principalmente dos tipos de tecnologías: las propietarias y las públicas. Mientras que el primer tipo pueden ser soluciones sencillas y robustas, tienen el grave inconveniente de ser difícilmente escalables y de forzar al cliente a permanecer con un único fabricante, con las limitaciones que eso supone. Por el contrario, las soluciones públicas permitirán la creación de redes heterogéneas muy escalables, ya que los distintos fabricantes crearán soluciones diferentes basadas en el mismo protocolo de comunicaciones acordado.

De este modo, actualmente se pueden encontrar en el mercado diversas tecnologías WSN propietarias como: Z-Wave, EnOcean, ANT, Wibree (ahora Bluetooth ULP), MiWi, etc.

Las principales diferencias entre ellas se pueden observar en la siguiente tabla:

Protocolo	Z-Wave	EnOcean	ANT	Bluetooth ULP	MiWi
Alcance	1 – 100m	100 - 300 m	1 – 30 m	1 - 10 m	1 – 100 m
Banda de Frecuencias	868.42 (Europa) 908.42 MHz (USA)	868.3 MHz o 315 MHz	2.4 GHz	2,4 GHz	2,4 GHz
Tasa de Transferencia	9.6 Kbps o 40 Kbps	125 Kbps (no mantenido)	1 Mbps	1 Mbps	250 Kbps
Modulación	GFSK	ASK	GFSK	GFSK	O-QPSK
Tamaño de Pila de Protocolo	16 KB – 26 KB	< 32 KB	2 KB	90 KB	3 KB – 17 KB

Características generales de las tecnologías WSN propietarias

Aunque a primera vista unas tecnologías parecen claramente mejores que otras en realidad cada una tiene su propio campo de aplicación. Por lo general, un mayor alcance y velocidades de transmisión suele implicar un mayor consumo (pero por ejemplo EnOcean ni siquiera

requiere baterías). Por otro lado, cuanta menos memoria requiera la pila de protocolos, ésta ofrecerá unas peores prestaciones (número máximo de nodos en red menor, incapacidad de formar redes malladas, etcétera) pero empleará probablemente chips más baratos.

Desafortunadamente, aunque muchas de estas tecnologías son muy eficientes, su campo de aplicación es muy reducido (por ejemplo Z-Wave se ha diseñado específicamente para automatización del hogar y Bluetooth ULP para redes muy pequeñas no malladas) y lo que más grave: no otorgan interoperabilidad entre dispositivos de diferentes fabricantes al tratarse de protocolos propietarios (a excepción de MiWi, cuyo protocolo es abierto pero requiere del uso de transceptores de MicroChip).

Es por ello que si se desea disponer de una solución interoperable y escalable, es imprescindible elegir una tecnología abierta. Esta es la razón de la importancia de las entidades de estandarización (IEEE por ejemplo) y las alianzas de empresas del sector, como la *ZigBee Alliance* [9].

De este modo, se dispone también de un número considerable de tecnologías abiertas basadas en especificaciones o estándares como: ZigBee, Wi-Fi, Bluetooth, Wireless USB, etc.

Estándar	**ZigBee**	**Wi-Fi (802.11 b y g)**	**Wi-Fi (802.11 b y g)**	**Wireless USB**
Alcance	1 – 100 m	1 – 250 m	1 – 250 m	1 – 10 m
Banda de Frecuencias	868.42 (Europa) 908.42 MHz (USA) 2.4 GHz (Global)	2.4 GHz	2.4 GHz	3.1 GHz – 10.6 GHz
Tasa de Transferencia	20 Kbps (Europa) 40 Kbps (USA) 250 Kbps (Global)	11 – 54 Mbps	11 – 54 Mbps	53 - 480 Mbps
Modulación	O-QPSK	OFDM	OFDM	MB-OFDM
Tamaño de Pila de Protocolo	40 KB – 100 KB	1000 KB	1000 KB	-

Características generales de algunas tecnologías WSN públicas

Una vez más cada una de las tecnologías ha sido diseñada con un diferente objetivo en mente, sin embargo su penetración en el mercado será mayor al permitir a cualquier fabricante desarrollar productos interoperables ciñéndose a una determinada especificación o estándar.

De entre todos ellos, resulta especialmente interesante la especificación **ZigBee**, basada en el estándar público **IEEE 802.15.4** y la alianza de compañías *ZigBee Alliance* [9] capaz de generar redes malladas muy seguras de hasta 65000 nodos, empleando dispositivos de muy bajo coste y consumo. ZigBee será pues la tecnología escogida para el desarrollo de este proyecto y se estudiará en profundidad en el apartado 2.3.

2.2. El estándar IEEE 802.15.4

2.2.1. Introducción y Aspectos Generales

El **IEEE 802.15.4** [1] [2] [3] es un estándar público del organismo internacional IEEE que definirá los niveles físico (PHY) y de acceso al medio (MAC) para las comunicaciones realizadas dentro de las redes inalámbricas personales de baja tasa de transmisión (LR-WPANs).

Este tipo de redes se emplea habitualmente para el intercambio de información en distancias relativamente cortas, requiriendo para ello de muy poca o ninguna infraestructura, lo cual posibilita el desarrollo de soluciones baratas, de tamaño reducido y muy eficientes en términos de consumo.

De este modo, el estándar se ha diseñado para proveer de conectividad inalámbrica (robusta, flexible y segura) a dispositivos móviles o fijos, de baja potencia y muy reducido consumo (alimentados habitualmente mediante baterías) y alcance limitado. No obstante, dependiendo de la aplicación y el tipo de dispositivo el estándar permitirá disponer de mayor cobertura a costa de una menor tasa de transferencia.

Las principales características del estándar serán pues:

1. Flujos binarios inalámbricos de 20 Kbps, 40 Kbps, 110 Kbps, 250 Kbps o 851 Kbps dependiendo del rango de frecuencias que se emplee.
2. Topologías de red en estrella o *peer-to-peer*.
3. Empleo de direcciones cortas (16 bits) o extendidas (64 bits).
4. Acceso al canal compartido mediante algoritmos de contienda (CSMA-CA o ALOHA, este último sólo para UWB), o libres de contienda mediante asignación de *slots* de tiempo (GTSs).
5. Soporte de asentimientos de tramas para mayor robustez de las comunicaciones.
6. Bajo consumo de energía.
7. Capacidad de detectar el nivel de energía (ED) y calidad del enlace de los canales (LQI).
8. Disponibilidad de 16 canales en la banda de 2450 MHz, 30 en la banda de 915MHz y 3 en la banda de 868MHz empleando técnicas de espectro ensanchado de secuencia directa (DSSS) y secuencia paralela (PSSS). Adicionalmente también se dispondrá de 14 canales de espectro ensanchado de chirps solapados (CSS) en la banda de 2,4GHz y 16 canales UWB (con sus frecuencias centrales en 500MHz y de 3,1 GHz hasta 10,6 GHz).

El estándar definirá dos tipos de dispositivos:

1. Dispositivos de Funcionalidad Completa (*Full Function Device*, FFD)
2. Dispositivos de Funcionalidad Reducida (*Reduced Function Device*, RFD)

Los cuales difieren en su uso y en la cantidad del estándar que implementan, ocupando así más o menos memoria según la aplicación deseada.

Un FFD puede funcionar como coordinador de la PAN (*Personal Area Network*), como coordinador (haciendo las veces de *router*) o como dispositivo final. Así mismo pueden comunicarse con cualquiera de estos tres tipos de dispositivos. Normalmente serán dispositivos dotados de más memoria, de mayor capacidad de cómputo o incluso de baterías de mayor tamaño (para una mayor autonomía).

Un RFD en cambio suele ser un dispositivo más simple, con menor capacidad para realizar tareas complejas o de larga duración. Habitualmente desempeñarán aplicaciones sencillas que consuman poca energía ya que típicamente estarán dotados de recursos escasos en términos de capacidad de cómputo o memoria disponible. Los RFDs sólo podrán comunicarse con un FFD, no pudiéndolo hacer con otros RFD, por lo que se comportarán siempre como dispositivos finales dentro de una red.

El estándar IEEE 802.15.4 fue publicado por primera vez en octubre del año 2003 [1] y ha sido revisado en dos ocasiones. La primera revisión, realizada en septiembre del año 2006 [2], dio lugar a una nueva versión del estándar. En cambio, la siguiente revisión efectuada en agosto del año 2007 [3] se limitó a corregir y completar algunos aspectos de la última versión publicada (incluyendo principalmente nuevos canales y técnicas de espectro ensanchado en el nivel físico). En este capítulo se describirán las tres versiones simultáneamente.

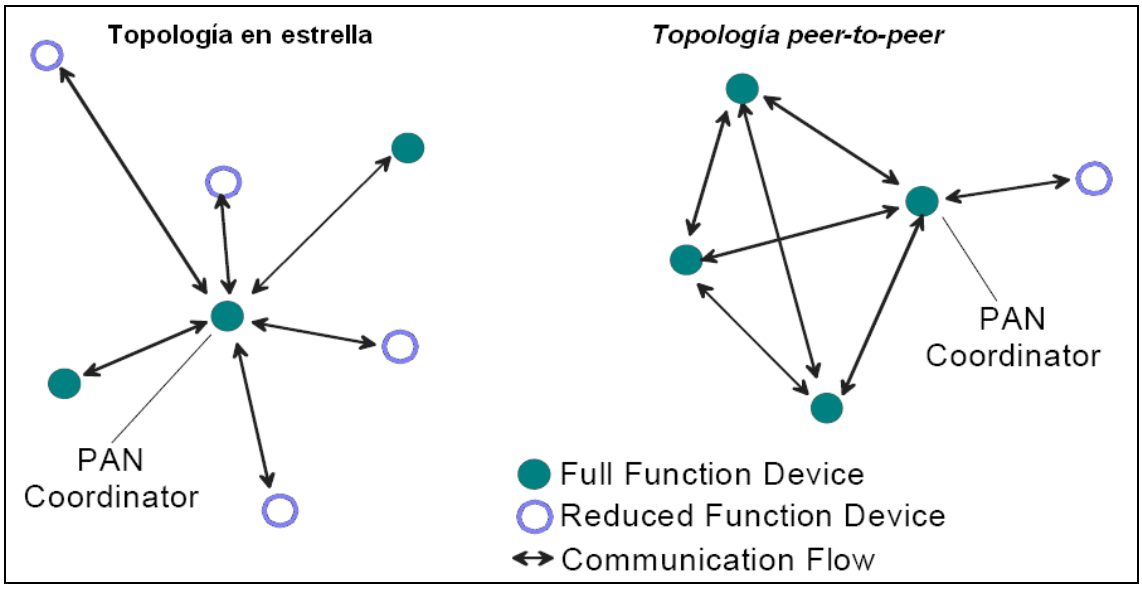
2.2.2. Topologías de Red

Dependiendo de los requisitos de cada aplicación, una red IEEE 802.15.4 puede implementar una de las siguientes topologías: topología en estrella y topología *peer-to-peer*.

Un requisito fundamental de ambas topologías será que todos los dispositivos de la red deberán disponer de una dirección extendida de 64 bits (única y de fábrica) que les permitirá identificarse de forma unívoca.

Adicionalmente todos los dispositivos obtendrán una dirección corta (de 16 bits) en el momento de la asociación a la red, que les permitirá realizar comunicaciones de forma más eficiente que empleando la dirección de 64 bits.

La siguiente figura ilustra la forma de las dos topologías de red mencionadas:



Topologías de una red 802.15.4 [2]

Cada PAN (*Personal Area Network*) individual dispondrá de un identificador único que permitirá a los dispositivos comunicarse dentro de la red o entre redes independientes (empleando para ello el identificador de la PAN en conjunción con sus direcciones corta o larga).

El mecanismo mediante el cual se eligen los identificadores (direcciones cortas y de PAN) queda fuera de las especificaciones del estándar 802.15.4. De igual forma, la definición del proceso de formación de redes en forma de estrella o *peer-to-peer* tendrá lugar en la especificación de la capa de red (fuera de este estándar), no obstante se introducirá brevemente dicho proceso a continuación.

2.2.2.1. Topología en estrella

En la topología de estrella, las comunicaciones se establecerán únicamente entre los dispositivos y un único controlador central llamado coordinador de la PAN (*PAN Coordinator*), que será el encargado de asignar las direcciones cortas de los dispositivos en el momento de su asociación a la red.

Habitualmente, cada dispositivo de la red dispondrá de una aplicación asociada determinada y será la fuente o el sumidero de diversas comunicaciones. Sin embargo, el coordinador de la PAN será adicionalmente el controlador principal de la red y será también capaz de iniciarla, terminarla o encaminar paquetes a través de ella.

Debido a estas responsabilidades añadidas, será habitual que el coordinador de la PAN disponga de más memoria y capacidad de proceso, además de alimentación vía red eléctrica mientras que la mayoría del resto de dispositivos de la red emplearán baterías.

Las redes en estrella suelen establecerse cuando un FFD se enciende por primera vez y trata de convertirse en el coordinador PAN de su propia red.

Para ello escogerá un identificador de PAN que no esté empleando ninguna red vecina, generará la nueva estrella y a continuación permitirá a otros dispositivos (RFDs o FFDs) asociarse a ella.

Las aplicaciones típicas de este tipo de topología son: automatización del hogar, periféricos de PC, juegos, juguetes y aplicaciones del cuidado personal de la salud.

2.2.2.2. Topología *peer-to-peer*

En una red con la topología peer-to-peer cualquier dispositivo podrá comunicarse con cualquier otro siempre que esté en su rango de cobertura y que no se trate de dos RFD.

Este tipo de redes también dispondrán de un único coordinador de la PAN, que habitualmente será el primer dispositivo FFD en iniciar una comunicación sobre el canal.

En esta ocasión los dispositivos se asociarán a la red a través de cualquier FFD dentro de su rango de cobertura (no necesariamente el coordinador de la PAN), de modo que la asignación de la dirección corta puede no depende necesariamente de un único nodo central.

Esta topología permite crear redes mucho más complejas que la topología en estrella, posibilitando por ejemplo la creación de redes malladas (*mesh*), en las que cada dispositivo de la red puede comunicarse con todos los demás o la creación de redes *cluster tree*, en la que extiende la cobertura al desplegar muchos FFD actuando como coordinadores y algunos RFD actuando como dispositivos finales.

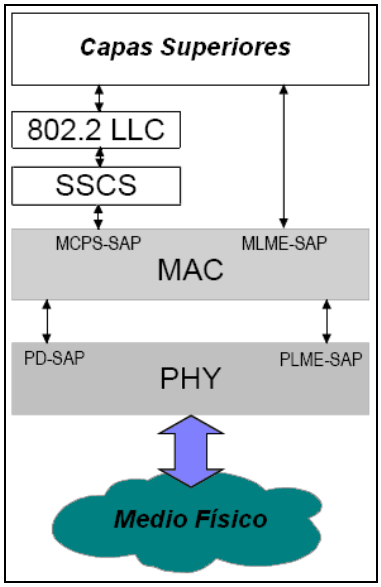
Este tipo de redes pueden ser *ad hoc*, auto organizativas y auto regenerativas. Adicionalmente permitirán encaminar paquetes a través de múltiples saltos entre dos dispositivos alejados dentro de la red (aunque esto dependerá del nivel de red elegido).

Las aplicaciones típicas de este tipo de topología serán: control y monitorización industrial, redes de sensores inalámbricos, gestión de inventarios, agricultura inteligente y seguridad.

2.2.3. Arquitectura de Red

La arquitectura del IEEE 802.15.4 se define en función de una serie de bloques destinados a simplificar el estándar, denominados capas o niveles. Cada nivel es responsable de una parte del estándar y ofrece unos determinados servicios a las capas superiores. El diseño de los diferentes niveles está basado en el modelo OSI de siete capas.

Una LR-WPAN se compone de los niveles físico (PHY) y de acceso al medio (MAC). Ambas capas dispondrán de diversos puntos de acceso (SAP) que actuarán de interfaz entre los diferentes niveles y permitirán definir los enlaces lógicos que se describirán a lo largo del estándar. La figura siguiente mostrará pues la arquitectura descrita:



Arquitectura de un dispositivo LR-WPAN

El nivel PHY involucrará al transmisor-receptor de radiofrecuencia junto a los mecanismos de bajo nivel que lo controlan. Proveerá de dos servicios: el servicio de datos (PD), que permite la transmisión y recepción de unidades de datos (PPDUs), y el servicio de gestión, que actuará de interfaz entre la MAC y la entidad de gestión de la capa física (PLME)).

El nivel MAC proporcionará acceso al canal físico para cualquier tipo de transferencia de información. De nuevo, pueden distinguirse dos servicios: el servicio de datos, que permite la transmisión y recepción de MPDUs (*MAC Protocol Data Units*) y el servicio de gestión (MLME).

Las capas superiores representadas en la figura anterior se refieren a una capa de red (que proporcionará las funcionalidades de configuración de red y manipulación y encaminamiento de mensajes), y a una capa de aplicación (que soportará la funcionalidad buscada para el dispositivo). Dichas capas superiores se encuentran fuera del alcance del estándar 802.15.4 y, a efectos de este proyecto, las definirá la especificación ZigBee (que se describirá en el apartado 2.3).

2.2.4. Nivel Físico (Capa PHY)

El nivel físico (PHY) corresponderá al nivel más bajo de la arquitectura de protocolos definida en el estándar IEEE 802.15.4, y se encargara de controlar directamente la transmisión de los datos por el medio. Esta capa será pues la responsable de:

1. Activar/desactivar el transceptor de radio.
2. Realizar la detección de energía (ED) dentro del canal seleccionado.
3. Dar una indicación de calidad del enlace (LQI) a partir de los paquetes recibidos.
4. Evaluar la disponibilidad del canal (CCA) para CSMA-CA.
5. Seleccionar la frecuencia de canal.
6. Gestionar la transmisión y recepción de datos por el canal.

A continuación se resumirán sus características más importantes.

2.2.4.1. Bandas de Frecuencia y Consideraciones Generales

Los canales de una LR-WPAN se situarán en las siguientes bandas de frecuencia sin licencia:

- 868 – 868.6 Mhz (Europa)
- 902 – 928 Mhz (EEUU)
- 2400 – 2483.5 (Mundial)
- 3100 – 10600 (variable dependiendo del país)

Podemos resumir las características de las diferentes PHY en la siguiente tabla:

PHY	Banda (Mhz)	Tasa chip (Kchip/s)	Modulación	Tasa bit (Kb/s)	Tasa Símbolo (Ksymbol/s)	Símbolos
868/915	868-868.6	300	BPSK	20	20	binario
868/915	902-928	600	BPSK	40	40	binario
868/915 (opc)	868-868.6	400	ASK	250	12.5	20 bit PSSS
868/915 (opc)	902-928	1600	ASK	250	50	5 bit PSSS
868/915 (opc)	868-868.6	400	O-QPSK	100	62.5	16-ario ort.
868/915 (opc)	902-928	1000	O-QPSK	250	62.5	16-ario ort.
2450 DSSS	2400 -2483.5	2000	O-QPSK	250	62.5	16-ario ort.
UWB sg (opc)	250-750					
2450 CSS (opc)	2400-2483.5		DQPSK	250	166.667	8-ario ort.
2450 CSS (opc)	2400-2483.5		DQPSK	1000	166.667	64-ario ort.

UWB bb (opc)	3244 – 4742					
UWB ba (opc)	5944 -10234					

Bandas de frecuencia y tasas de bit de la capa PHY

Como puede apreciarse en la tabla, las bandas de 868/915 MHz con modulaciones BPSK y O-QPSK emplearán espectro ensanchado DSSS mientras que aquellas con modulación ASK, emplearán espectro ensanchado por secuencia paralela (PSSS).

La revisión del estándar del 2007 incorporó dos nuevas configuraciones posibles al nivel físico (que corresponderán a las 5 últimas entradas de la tabla).

La primera de ellas consistirá en emplear técnicas de espectro ensanchado por *chirps* (CSS) en la banda de 2450 MHz, junto con una modulación DQPSK de constelaciones 8-arias o 64-arias. La segunda configuración implicará emplear una versión de espectro ensanchado UWB (*Ultra Wideband*) sobre nuevas bandas de frecuencia.

Ambas configuraciones presentarán la ventaja de que proporcionarán resistencia adicional a los desvanecimientos multitrayecto que tienen lugar con bajas potencias de transmisión.

De este modo, como ya se indicó con anterioridad, el nivel físico definirá las transmisiones inalámbricas para 16 canales en la banda de 2450 MHz, 30 en la banda de 915 MHz y 3 en la banda de 868 MHz, empleando técnicas de espectro ensanchado de secuencia directa (DSSS) y secuencia paralela (PSSS). Adicionalmente también definirá 14 canales de espectro ensanchado de chirps solapados (CSS) en la banda de 2,4GHz y 16 canales UWB (con sus frecuencias centrales en 500MHz y de 3,1 GHz hasta 10,6 GHz).

En cuanto a consideraciones de sensibilidad se refiere, un dispositivo conforme al estándar funcionando en las bandas de 868 o 915 MHz empleando una modulación BPSK requerirá de una sensibilidad mínima de -92 dBm, mientras que si se emplean modulaciones O-QPSK o ASK dicha sensibilidad mínima será de -85 dBm.

La sensibilidad mínima requerida en la banda de 2,4GHz será de -85 dBm si se emplean técnicas de espectro ensanchado DSSS o CSS a una tasa de bit de 1 Mbps. Si se emplea en cambio CSS con una tasa de 250 Kbps, será necesaria una sensibilidad mínima de -91 dBm.

En cuanto a potencia se refiere, un dispositivo transmisor funcionando bajo las especificaciones del 802.15.4 deberá ser capaz de transmitir -3 dBm de potencia y de recibir al menos un máximo de -20 dBm de señal deseada. Sin embargo, los máximos de potencia transmitibles en un canal concreto dependerán de la legislación de cada región.

2.2.4.2. Formato de trama

La estructura de la trama PPDU (*PHY Protocol Data Unit*), dependiendo de la banda de frecuencias y la técnica de espectro ensanchado elegida, se presentará en las siguientes figuras:

		Octetos		
		1		Variable
Preámbulo	SFD	Longitud de trama (7 bits)	Reservado (1 bit)	PSDU
SHR		PHR		Datos

Formato de la trama PPDU (excepto para UWB y CSS)

		Bits	Octetos
		19	Variable
Preámbulo	SFD	PHR	PSDU
SHR		PHR	Datos

Formato de la trama PPDU (para UWB)

Tasa de datos	SHR		PHR	PSDU
	Preámbulo	SFD		
1 Mb/s	8 símbolos	4 símbolos	2 símbolos	Variable
250 Kb/s (opcional)	20 símbolos	4 símbolos	8 símbolos	Variable

Formato de la trama PPDU (para CSS)

De este modo, una trama PPDU se compone de los siguientes bloques:

- **SHR** (*Synchronization Header*): cabecera que permitirá al dispositivo sincronizarse para una correcta diferenciación de los distintos bits del mensaje.
- **PHR** (*PHY Header*): cabecera real del mensaje que contendrá información sobre la longitud de la trama transmitida. En el caso de UWB, también incluirá información acerca de la tasa, el *ranking* y el preámbulo.
- **Datos**: campo de longitud variable que transportará la trama del subnivel MAC.

Dentro de la SHR, el campo "preámbulo" se empleará por el transmisor-receptor para sincronizarse a nivel de chip y a nivel de símbolo con el mensaje entrante. El campo SFD (*Start of Frame Delimiter*) será el delimitador de inicio de trama e indicará dónde acaba el preámbulo y donde comienza la cabecera de la trama en sí. Ambos campos presentarán un formato diferente en función del nivel físico empleado.

Dentro de la PHR, el campo de "longitud de trama" servirá principalmente para indicar la longitud exacta de los datos que vienen a continuación, que puede ser variable.

Finalmente, el campo de "datos" transportará la trama de nivel superior (MAC).

2.2.4.3. Servicios del nivel físico

El nivel físico provee de dos servicios hacia la capa superior (la MAC): el servicio de datos (PD), que permitirá la transmisión y recepción PPDUs, y el servicio de gestión (PLME).

Estos dos servicios se gestionarán a través de un conjunto de primitivas. Para el caso del servicio de datos se dispone de las siguientes:

Primitiva	Tipo 1	Tipo 2	Tipo 3
PD-DATA	Request	Confirm	Indication

Primitivas del servicio de datos capa PHY

Estas tres primitivas **PD-DATA** se encargarán de solicitar el envío de datos por el canal (*.request*), confirmar el envío o error de estos datos a la capa MAC (*.confirm*) y de indicar a la capa MAC del dispositivo receptor de la llegada de éstos (*.indication*).

Para el caso del servicio de gestión se dispondrá de las siguientes primitivas:

Primitiva	Tipo 1	Tipo 2	Tipo 3
PLME-CCA	Request	Confirm	--
PLME-ED	Request	Confirm	--
PLME-GET	Request	Confirm	--
PLME-SET-TRX-STATE	Request	Confirm	--
PLME-SET	Request	Confirm	--
PLME-DPS	Request	Confirm	Indication
PLME-SOUNDING	Request	Confirm	--
PLME-CALIBRATE	Request	Confirm	--

Primitivas del servicio de gestión capa PHY

Las primitivas **PLME-CCA** se encargarán de solicitar una evaluación de la disponibilidad del canal para enviar datos (*.request*) y devolverán los resultados de esta evaluación a la capa MAC (*.confirm*).

Las primitivas **PLME-ED** solicitarán (*.request*) la detección de energía de un canal y devolverán esta información a la capa superior (*.confirm*).

Las primitivas **PLME-GET** y **PLME-SET** solicitarán (*.request*) el envío a la capa superior del valor de una variable de entorno de la capa física (**GET**) o cambiarán su valor al especificado por la MAC (**SET**). El resultado de estas operaciones se reportará a la capa MAC por la primitiva de confirmación (*.confirm*) que corresponda.

Las primitivas **PLME-SET-TRX-STATE** se encargarán de solicitar (*.request*) a la capa física que cambie el estado de operación del transceptor de radio a una de las tres posibles situaciones: Transceptor apagado (*TRX_OFF*, con el menor consumo de potencia), Transmisor encendido (*TX_ON*) o Receptor encendido (*RX_ON*). El resultado de esta operación se comunicará a la capa superior a través de la primitiva de confirmación (*.confirm*).

Las primitivas **PLME-DPS**, **PLME-SOUNDING** y **PLME-CALIBRATE** son opcionales y sólo se emplearán en UWB en procesos de medición de distancia (*ranging*).

Cada una de estas primitivas tendrá una serie de parámetros que la capa superior ajustará según los parámetros de la primitiva de nivel superior que reciba. Puesto que el fabricante del chip elegido suministrará las librerías precompiladas de todo el nivel físico, no será necesario profundizar más en estas primitivas para el desarrollo del proyecto.

2.2.5. Nivel de Acceso al Medio (Capa MAC)

Esta capa de la arquitectura gestiona todos los accesos al canal radio físico y es responsable de:

1. Generar balizas (*beacons*) si el dispositivo es un coordinador
2. Sincronizarse a las balizas
3. Soportar asociación y desasociación con la PAN
4. Soportar seguridad
5. Gestionar el uso de los mecanismos de acceso al medio con contienda (CSMA-CA o ALOHA en el caso de disponer de una PHY de UWB) y libres de contienda
6. Gestión y mantenimiento del mecanismo de GTS
7. Proveer de un canal fiable entre dos entidades MAC extremo a extremo

A continuación se describirán los puntos más importantes del funcionamiento de esta capa.

2.2.5.1. Mecanismos de Control de Acceso al Medio

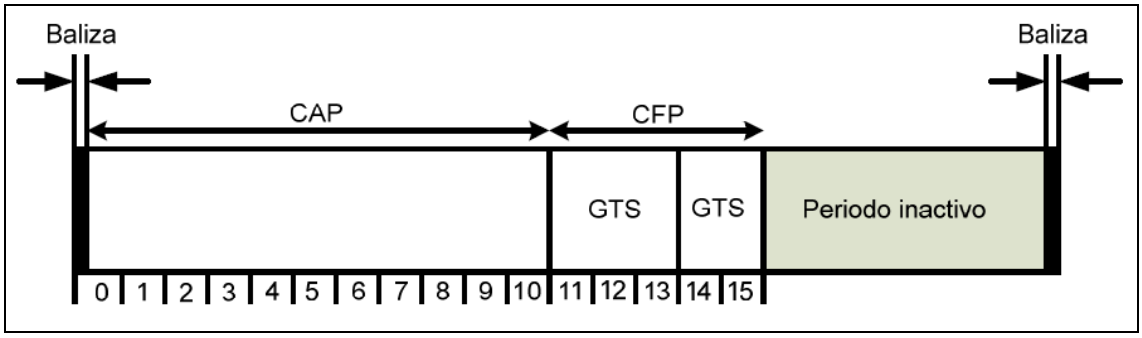
El estándar 802.15.4 permite la utilización de una estructura de supertrama que define el coordinador de la PAN para el acceso al medio compartido.

Las supertramas están delimitadas por el envío de balizas (*beacon*s) y pueden tener una región activa y una región inactiva. Durante la región activa el coordinador fijará según sus necesidades otras dos regiones: región de acceso por contienda (*Contention Access Period*,

CAP) y región libre de contienda (*Contention Free Period*, **CFP**). Durante la región inactiva se prohibirá el envío de tramas y el coordinador de la PAN podrá entrar en bajo consumo o dedicarse a tareas de gestión.

En la **CAP** los dispositivos accederán al medio mediante contienda por medio de CSMA-CA ranurado mientras que en la **CFP** (si existe esta región, puesto que es configurable) se crean "canales" virtuales en las que el coordinador concede acceso al medio sólo a un dispositivo. Estos "canales" se conocen como **GTS** (*Guaranteed Time Slot*) y la información acerca de su tamaño y dispositivo asociado se enviará en cada baliza.

De utilizarse supertramas, el coordinador dividirá la zona activa de cada una en 16 ranuras (*slots*) iguales de tiempo y transmitirá una baliza (*beacon frame*) en el primer *slot* de forma que todos los dispositivos asociados a ese coordinador la escucharán y se sincronizarán a ésta tal como aparece en la figura siguiente:



Estructura de una supertrama

La utilización de balizas (y así estructura de supertramas) es opcional y dependerá de los objetivos de diseño a optimizar tipo de red a implementar. Así mismo, en el caso de utilizar supertramas, la utilización de slots **GTS** también es una opción de diseño.

El nivel MAC necesita un tiempo determinado para procesar los datos que le llegan del nivel físico. Por lo tanto, las tramas enviadas desde un dispositivo deben estar separadas al menos un periodo de tiempo llamado **IFS**. Si una primera transmisión exige un asentimiento, la separación temporal entre dicho asentimiento y la segunda transmisión deberá ser también de al menos un periodo **IFS**.

La duración de este periodo **IFS** dependerá del tamaño de la trama recibida, distinguiéndose así un periodo corto (**SIFS**, asociado a tramas recibidas cortas) y otro largo (**LIFS**, asociado a tramas recibidas largas).

Si se prescinde de balizas, los dispositivos enviarán datos al coordinador mediante CSMA-CA no ranurado y éste responderá opcionalmente con tramas de asentimiento (ACK). El envío de datos desde un coordinador a un dispositivo final (RFD) se realizará mediante CSMA-CA no

ranurado de forma indirecta: el dispositivo solicitará datos al coordinador, éste asentirá esta solicitud y enviará los datos y, finalmente, el dispositivo asentirá el envío.

Si se trabaja con balizas (y por tanto supertramas), el envío de datos por parte del dispositivo final al coordinador se realizará de la misma forma que en un entorno sin balizas con la diferencia de que el dispositivo se sincronizará primero con el coordinador y enviará los datos mediante CSMA-CA ranurado durante la **CAP**. El envío de datos desde el coordinador hacia el dispositivo final asociado será exactamente igual que sin balizas con la diferencia de que el dispositivo sabrá si hay mensajes pendientes para él al sincronizarse y leer las balizas.

En las ocasiones en que se emplee CSMA-CA no ranurado, el dispositivo que desee enviar deberá esperar primero un tiempo aleatorio. Una vez haya expirado, se evaluará si el canal está libre realizando un CCA (*Clear Channel Assesment*), en cuyo caso la transmisión se realizará inmediatamente.

En caso contrario, el dispositivo deberá volver a esperar otra cantidad de tiempo aleatorio antes de realizar el siguiente intento. Transcurrido un número máximo de intentos sin éxito, se dará por finalizado el proceso y se considerará que ha habido un fallo en la transmisión.

El funcionamiento del CSMA-CA ranurado es muy similar a la versión no ranurada, con la salvedad de que en lugar de esperar periodos de tiempo aleatorios, será necesario esperar un número de slots aleatorio.

2.2.5.2. Modelos de Transferencia de datos

El estándar IEEE 802.15.4 contempla tres escenarios distintos de intercambio de datos:

- Transmisión de datos de un coordinador a un dispositivo.
- Transmisión de datos de un dispositivo a un coordinador.
- Transmisión de datos entre dos dispositivos iguales (*peers*)

A continuación, se describirá resumidamente el funcionamiento de cada uno, diferenciando los casos de redes con y sin balizas.

Transmisión de datos de un coordinador a un dispositivo

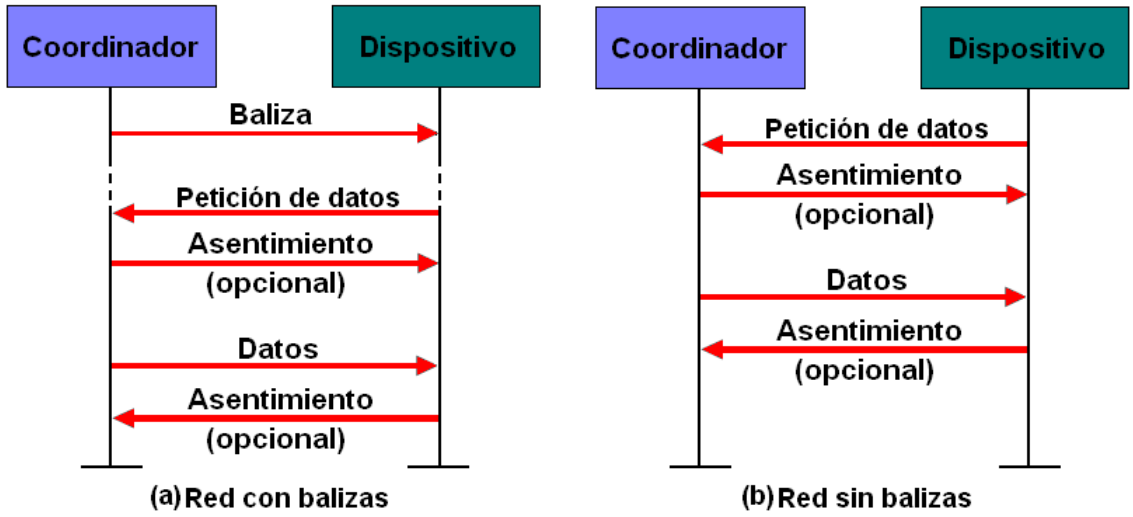
En este escenario suele ser habitual que el dispositivo destino de la transmisión se encuentre en letargo (para ahorrar baterías), de forma que el coordinador no podrá enviarle los datos directamente y tendrá que almacenarlos hasta que el dispositivo los solicite.

Si este escenario tiene lugar en una red con balizas, el dispositivo estará sincronizado con éstas y saldrá periódicamente de su letargo para escucharlas.

De este modo, el coordinador anunciará en cada baliza un listado de los dispositivos para los que dispone datos almacenados y éstos procederán a solicitarlos mediante la emisión de un comando MAC de petición de datos, empleando CSMA-CA ranurado.

Esta trama de petición de datos se asentirá opcionalmente por el coordinador y a continuación procederá a transmitir los datos solicitados mediante CSMA-CA ranurado al dispositivo destino que, opcionalmente, también asentirá su recepción.

Todo el proceso descrito se ilustra en el caso (a) de la figura siguiente:



Escenarios de envío de datos de un coordinador a un dispositivo

Si la transmisión tiene lugar en una red sin balizas, el dispositivo deberá solicitar periódicamente datos del coordinador mediante la emisión de tramas MAC de petición de datos, empleando CSMA-CA no ranurado.

El coordinador asentirá opcionalmente dichas peticiones de datos y, en el caso de disponer de datos almacenados destinados al dispositivo en cuestión, procederá a transmitirlos mediante CSMA-CA no ranurado. El dispositivo receptor asentirá los datos opcionalmente.

En el caso de no disponer de datos almacenados destinados al dispositivo solicitante, el coordinador lo indicará en el asentimiento de la solicitud o mediante el envío de una trama vacía.

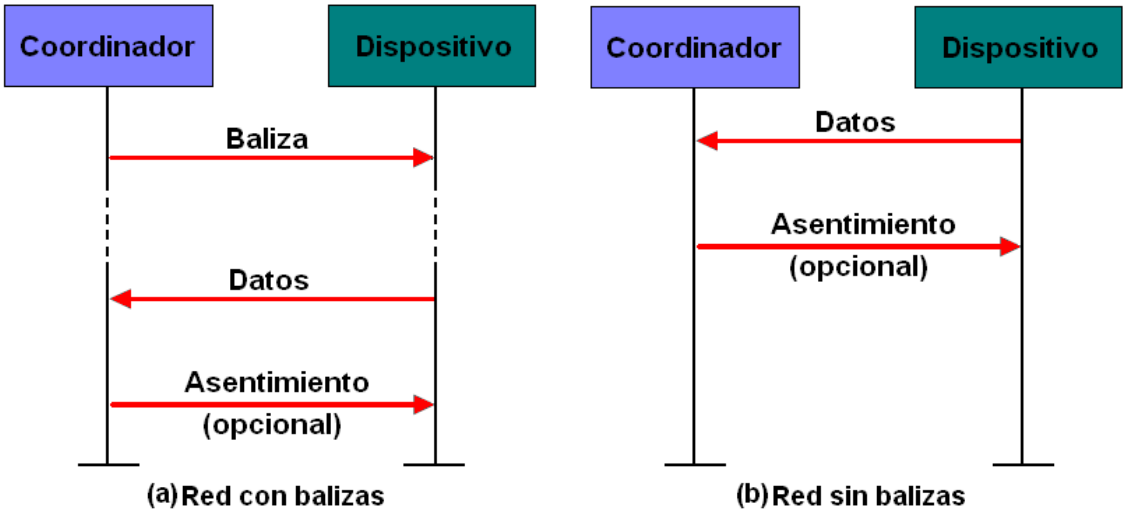
El proceso descrito para una red sin balizas se ilustra en el caso (b) de la figura anterior.

Transmisión de datos de dispositivo a un coordinador

En este escenario, cuando un dispositivo quiera enviar tramas a un coordinador en una red con balizas, le bastará con sincronizarse con la supertrama y transmitir los datos directamente mediante CSMA-CA ranurado. El coordinador a su vez los asentirá opcionalmente.

En el caso de que este escenario tenga lugar en una red sin balizas, el dispositivo transmitirá los datos directamente mediante CSMA-CA no ranurado y el coordinador los asentirá opcionalmente. En este tipo de redes no será necesaria una sincronización previa entre los dispositivos.

El proceso de transmisión de datos de este escenario se ilustra, para los casos de redes con y sin balizas, en la siguiente figura:



Escenarios de envío de datos de un dispositivo a un coordinador

Transmisión de datos *peer-to-peer*

En una PAN *peer-to-peer*, todo dispositivo puede comunicarse con cualquier otro dentro de su esfera de cobertura.

Para que esta transmisión de datos se realice con éxito, los dispositivos que deseen comunicarse deben estar sincronizados o continuamente enviándose información.

En el primer caso, para lograr la sincronización entre los dispositivos será necesario realizar una serie de medidas que el estándar 802.15.4 no especifica.

En el segundo caso, todo dispositivo podrá siempre transmitir datos a otro simplemente empleando CSMA-CA no ranurado.

2.2.5.3. Formato de Trama

El estándar IEEE 802.15.4 especifica cuatro tipos diferentes de tramas MAC (MPDUs): la trama baliza (*beacon*), la trama de datos, la trama de asentimiento (ACK) y la trama de comandos MAC.

Todas ellas constan de tres grandes bloques: la cabecera MAC (**MHR**), situada al comienzo de la trama, la carga de datos (***payload***), de longitud variable, y por último, la cola (**MFR**).

El **MFR** contendrá una secuencia de comprobación de trama (**FCS**) de 16 bits para corrección y detección de errores en la trama recibida, y tendrá la misma longitud en todas las tramas. El **MHR** en cambio, variará de tamaño, siendo menor en el caso de tratarse de una trama de asentimiento.

El formato general descrito puede apreciarse en la figura siguiente:

<table>
<tr><th>Octetos: 2</th><th>1</th><th>0/2</th><th>0/2/8</th><th>0/2</th><th>0/2/8</th><th>0/5/6/10/14</th><th>variable</th><th>2</th></tr>
<tr><td rowspan="2">Control de trama</td><td rowspan="2">Nº secuencia</td><td>ID PAN destino</td><td>Dir. destino</td><td>ID PAN origen</td><td>Dir. origen</td><td rowspan="2">Cabecera auxiliar de seguridad</td><td rowspan="2">Datos</td><td rowspan="2">FCS</td></tr>
<tr><td colspan="4">Campos de direccionamiento</td></tr>
<tr><td colspan="7">MHR</td><td>Payload</td><td>MFR</td></tr>
</table>

Formato general de trama MAC

A continuación se procederá a describir resumidamente cada uno de los campos de la trama:

El campo de **control de trama** estará presente independientemente del tipo de trama y contendrá información sobre el tipo de trama de que se trata, campos de direccionamiento incluidos y otros flags de control.

Bits: 0-2	3	4	5	6	7-9	10-11	12-13	14-15
Tipo de trama	Habilitación seguridad	Trama pendiente	Petición Ack	Compresión PAN ID	Reserv.	Modo Dir. Destino	Ver. trama	Modo Dir. Origen

Campo de "control de trama"

Como puede deducirse de la figura anterior, el campo de control de trama indicará al receptor exactamente el formato de los campos de la trama transmitida, para asegurar así su correcta interpretación en recepción. Adicionalmente indicará si quedan aún tramas por transmitir destinadas al mismo receptor, y si se solicita el envío de asentimientos de su recepción.

El **número de secuencia** contendrá el identificador de secuencia de la trama. Para una trama baliza este campo tendrá el valor de secuencia de trama baliza (BSN) y para el resto de tipos de trama, tendrá el número de secuencia de datos (DSN).

Los **campos de direccionamiento** incluirán, dependiendo del tipo de trama, el identificador de la PAN del dispositivo destino de la trama, su dirección, el identificador de la PAN del dispositivo origen y la dirección de este último.

La **cabecera auxiliar de seguridad** contendrá la información necesaria para los procesos relacionados con la seguridad, incluyendo cómo debe de protegerse la trama y qué claves deben ser usadas de la base de datos interna (PIB). Este campo sólo parecerá si se habilita la seguridad a nivel de MAC.

El campo **FCS** será la secuencia de control de trama que permitirá evaluar la integridad de la trama en recepción. El valor del campo se calculará mediante un código de redundancia cíclica (CRC) de 16 bits, a partir del **MHR** y del ***payload*** de la trama MAC.

Una vez definido el formato genérico de las tramas MAC, los cuatro tipos existentes se diferenciarán principalmente en el formato del payload de datos y en la inclusión/exclusión de determinados campos de la cabecera (**MHR**).

Puesto que no se requiere, a efectos de este proyecto, un análisis más concienzudo del formato de las tramas MAC, no se profundizará más en su estudio. Si se desea adquirir un mayor conocimiento sobre este campo, se recomienda la lectura del capítulo 7.2 de la bibliografía recomendada **[1]** y **[2]**.

2.2.5.4. Servicios del nivel MAC

El nivel de acceso al medio (MAC) provee de dos servicios a las capas superiores: el servicio de datos (MCPS), y el servicio de gestión, (MLME). Entre los dos formarán un interfaz entre la capa superior (SSCS) y el nivel físico (PHY), a través de sendos puntos de acceso (SAPs).

De igual forma que sucedía en el nivel PHY, los servicios se gestionarán a través de sus puntos de acceso mediante un conjunto de primitivas.

Así pues, el servicio de datos (MCPS), empleará las siguientes primitivas:

Primitiva	Tipo 1	Tipo 2	Tipo 3
MCPS-DATA	Request	Confirm	Indication
MCPS-PURGE	Request	Confirm	- -

Primitivas del servicio de datos MAC (MCPS)

Las primitivas **MCPS-DATA** permitirán a la capa superior solicitar a la MAC (.*request*) el envío de datos a un determinado destino. Como respuesta, la capa MAC generará a la capa superior una primitiva de confirmación con el resultado de la operación (.*confirm*).

En el caso de transmitirse el paquete a la capa física correctamente, ésta enviará los datos y la capa MAC destino generará una primitiva de indicación de llegada de datos a la capa superior (.indication). Esta primitiva llevará consigo los datos recibidos, información acerca de la calidad del enlace y opciones de seguridad, entre otras cosas.

Las primitivas **MCPS-PURGE** permitirán a la capa superior solicitar a la capa MAC (.*request*) la eliminación de un paquete MSDU (*Mac Service Data Unit*) de la cola de transacción. El resultado de esta operación se notificará a la capa superior mediante la primitiva de confirmación (.*confirm*).

De igual forma, el servicio de gestión de la capa MAC (MLME), empleará las siguientes primitivas:

Primitiva	Tipo 1	Tipo 2	Tipo 3	Tipo 4
MLME-ASSOCIATE	Request	Indication	Response	Confirm
MLME-DISASSOCIATE	Request	Indication	--	Confirm
MLME-BEACON-NOTIFY	--	Indication	--	--
MLME-GET	Request	--	--	Confirm
MLME-GTS	Request	Indication	--	Confirm
MLME-ORPHAN	--	Indication	Response	--
MLME-RESET	Request	--	--	Confirm
MLME-RX-ENABLE	Request	--	--	Confirm
MLME-SCAN	Request	--	--	Confirm
MLME-COMM-STATUS	--	Indication	--	--
MLME-SET	Request	--	--	Confirm
MLME-START	Request	--	--	Confirm
MLME-SYNC	Request	--	--	--
MLME-SYNC-LOSS	--	Indication	--	--
MLME-POLL	Request	--	--	Confirm

Primitivas del servicio de gestión MAC (MLME)

Las primitivas **MLME_ASSOCIATE** permitirán a la capa superior solicitar la asociación con un dispositivo coordinador (.*request*). Esta solicitud se transferirá al dispositivo destino, que estudiará si permitir dicha asociación y hará llegar su decisión a la capa superior del dispositivo

solicitante a través de una confirmación de la capa MAC (*.confirm*). Dicha confirmación incluirá también los errores que hayan tenido lugar, en caso de producirse alguno.

En el lado del coordinador, la capa MAC transferirá a la capa superior una indicación de la solicitud de asociación (*.indication*) y ésta decidirá si aceptarla o rechazarla. Su decisión se transmitirá desde la capa superior a la capa MAC (*.response*) y ésta se encargará de comunicársela al dispositivo solicitante mediante envío indirecto. Si tuviese lugar algún error a lo largo del procedimiento, éste se informaría a la capa superior del dispositivo coordinador mediante la primitiva **MLME-COMM-STATUS***.indication*.

Es necesario recordar que cada solicitud de la capa superior a la capa MAC se traducirá habitualmente en una secuencia de transmisiones y recepciones de paquetes de la capa física antes de obtenerse ninguna confirmación o indicación del proceso.

Las primitivas **MLME-DISASSOCIATE** permitirán a un dispositivo solicitar (*.request*) la desasociación con un coordinador o a un coordinador requerir la desasociación de un dispositivo. El resultado de la operación se comunicará luego a la capa superior mediante una primitiva de confirmación (*.confirm*).
Por otro lado, la capa MAC del destino de la petición de desasociación emitirá a su capa superior la indicación (*.indication*) de haber sido desasociado de la red (en caso de que el dispositivo solicitante sea un coordinador) o de la desasociación de un dispositivo (en caso de que el dispositivo solicitante no sea el coordinador).

La primitiva **MLME-BEACON-NOTIFY***.indication* se generará por la capa MAC de un dispositivo hacia su capa superior cuando reciba tramas baliza de un coordinador. Esta primitiva incluirá pues la información contenida en la baliza.

Las primitivas **MLME-GET** permitirán a la capa superior solicitar (*.request*) el valor de algún atributo de la capa MAC (de la PIB, la *PAN Information Base*). La información recuperada se transmitirá a la capa superior mediante la primitiva de confirmación (*.confirm*).

Las primitivas **MLME-SET** permitirán a la capa superior fijar (*.request*) el valor de algún atributo de la capa MAC (PIB). El resultado de esta operación se transmitirá a la capa superior mediante la primitiva de confirmación (*.confirm*).

Las primitivas **MLME-GTS** permitirán a la capa superior de los dispositivos solicitar (*.request*) la asignación o desasignación de un GTS con unas características determinadas. Una vez más la MAC transmitirá a su capa superior, mediante una primitiva de confirmación (*.confirm*), el éxito o fracaso del establecimiento del GTS.

Cuando la petición de asignación o desasignación sea concedida por un coordinador, la capa MAC de éste notificará a su capa superior este suceso (.*indication*). Esta primitiva también se generará en la capa MAC de un dispositivo al que un coordinador haya desasignado un GTS.

Las primitivas **MLME-ORPHAN** permitirán indicar (.*indication*) a la capa superior de un coordinador de que ha recibido una notificación de dispositivo huérfano. La capa superior evaluará pues si ese dispositivo estaba asociado al coordinador y generará una primitiva de respuesta (.*response*) a la capa MAC con la información al respecto. Si el dispositivo estaba asociado entonces la capa MAC generará un comando de realineación con el dispositivo y se reasociará. Si no estaba asociado se ignorará esta primitiva.
El éxito o fracaso del proceso de realineación con el dispositivo se comunicará a la capa superior del coordinador mediante la primitiva **MLME-COMM-STATUS**.*Indication*.

Las primitivas **MLME-RESET** permitirán a la capa superior de un dispositivo reiniciar las variables internas de su capa MAC a su valor por defecto manteniendo (o no) el valor de los atributos del PIB. El éxito o fracaso de esta solicitud (.*request*) se comunicará a la capa superior a través de la primitiva de confirmación (.*confirm*).

Las primitivas de **MLME-RX-ENABLE** permitirán a la capa superior solicitar que el receptor esté encendido durante un periodo finito de tiempo o desactivado. La petición (.*request*) provocará la generación de un considerable número de primitivas de capa física y el uso de temporizadores. El éxito o fracaso de esta operación (y las causas de ello) se informarán a la capa superior a través de la primitiva de confirmación (.*confirm*).

Las primitivas **MLME-SCAN** permitirán a la capa superior solicitar (.*request*) un escaneo de la energía de una lista dada de canales. Esto permitirá a la capa superior evaluar la energía que se encuentra localizada en éstos, buscar coordinadores PAN (el propio u otros), balizas, etc.

El escaneo puede ser de cuatro formas: **detección de energía** (*Energy Detection*, ED), **escaneo activo**, **escaneo pasivo** o **escaneo de huérfanos** (*Orphan Scan*).

El resultado de esta operación (lista de descriptores de PAN o listas de niveles de energía), así como el éxito o fracaso de la misma, se comunicarán a la capa superior mediante la primitiva de confirmación (.*confirm*).

La primitiva **MLME-COMM-STATUS**.*indication* permitirá informar a la capa superior del estado de una comunicación. Esta primitiva se generará en muchas situaciones de error de comunicación o incidencias de seguridad, así como en el proceso (exitoso o no) de asociación de dispositivos o de realineamiento de dispositivos huérfanos.

Las primitivas **MLME-START** permitirán a un dispositivo FFD solicitar (.*request*) la creación de una nueva PAN o el empleo de una nueva configuración de supertrama.
El resultado se informará a la capa superior a través de la primitiva de confirmación (.*confirm*).

La primitiva **MLME-SYNC**.*request* permitirá a la capa superior solicitar sincronización del dispositivo con las balizas enviadas por un coordinador en un canal determinado. Opcionalmente esta solicitud permite especificar si se desea sincronizarse tan sólo con la primera baliza que se reciba o con todas las que lleguen además de la primera.

La primitiva **MLME-SYNC-LOSS**.*indication* se generará en la capa MAC de un dispositivo hacia la capa superior en caso de pérdida de sincronización con un coordinador, indicando adicionalmente las posibles causas.

Las primitivas **MLME-POLL** permitirán a la capa superior solicitar a un coordinador (.*request*) el envío de datos pendientes para el dispositivo. La capa MAC del dispositivo solicitante informará a su capa superior del resultado de la petición (.*confirm*) y, en caso de haber recibido datos pendientes, se transmitirán seguidamente mediante la primitiva correspondiente (.*indication*).

Todas las primitivas estudiadas permitirán estimar las prestaciones que las capas física y de acceso al medio de 802.15.4 pueden llegar ofrecer.

Para alcanzar un conocimiento en mayor profundidad acerca del exacto funcionamiento de los comandos enviados al medio así como de su formato, sistemas de seguridad, reenvío de tramas, comprobación de errores y demás detalles de bajo nivel, se recomienda acudir al estándar IEEE 802.15.4. En lo que a este proyecto se refiere, se trabajará con esos aspectos de forma transparente, a través de los parámetros contenidos en las primitivas ya descritas.

2.3. La especificación ZigBee

La especificación **ZigBee** es un estándar de facto, desarrollado por la *ZigBee Alliance* (asociación de decenas de empresas), en busca de la definición de un protocolo global y abierto para el intercambio de datos inalámbricos en sistemas de automatización de hogares y edificios, electrónica de consumo, control industrial, escenarios de monitorización y control, entornos socio sanitarios, periféricos de PC y juegos, etcétera, entre dispositivos de bajo coste y bajo consumo.

El objetivo principal de ZigBee será definir los niveles de red, seguridad y aplicación (modelo OSI de siete capas) en escenarios de redes inalámbricas de sensores, basándose en el nivel físico y de acceso al medio (MAC) definidos por el estándar IEEE 802.15.4.

La primera versión de la especificación vio la luz en diciembre del año 2004 [4] pero no llegó a extenderse en el mercado y a día de hoy se considera obsoleta.

La segunda versión de la especificación se publicó en octubre del año 2006 [5] y presenta considerables modificaciones con respecto a la versión original, siendo ambas incompatibles.

La tercera y última versión publicada (octubre del año 2007 [6]), permite dos perfiles distintos de pila (*stack*):

- Perfil de pila 1 (s*tack profile 1*): llamado simplemente **ZigBee**, diseñada para aplicaciones comerciales de control y monitorización del hogar, capaces de presentar *firmwares* más pequeños para dispositivos de capacidad reducida.
- Perfil de pila 2 (*Stack profile 2*): llamado **ZigBee PRO**, que ofrece nuevas funcionalidades como envíos *multicast*, agilidad de frecuencia, mecanismos de encaminamiento *many-to-one* (para la recolección centralizada de datos), alta seguridad con intercambio de claves simétricas (SKKE), mayor capacidad de fragmentación de paquetes, etcétera. Requiere de dispositivos con mayor capacidad de almacenamiento y memoria.

ZigBee 2007 es completamente compatible con ZigBee 2006, de modo que un dispositivo ZigBee 2007 podrá asociarse y funcionar correctamente en una red ZigBee 2006 y viceversa.

Por el contrario, debido a diferencias incompatibles en el algoritmo de encaminamiento, los dispositivos ZigBee 2006 o ZigBee 2007 sólo podrán funcionar en una red ZigBee PRO como dispositivos finales (ZEDs), del mismo modo que los dispositivos ZigBee PRO sólo podrán funcionar como dispositivos finales en una red ZigBee 2006 o ZigBee 2007.

Adicionalmente, se han publicado algunas revisiones de cada una de las versiones de la especificación, que se han limitado a incluir clarificaciones del texto y a corregir erratas.

Desafortunadamente, debido a la política interna de la *ZigBee Alliance*, ha existido una considerable diferencia temporal entre la publicación de cada especificación y su disponibilidad al público general (aquellos que no son miembros de la *ZigBee Alliance*). Este hecho, en conjunción con los retrasos sucedidos en la publicación de librerías por parte de los distintos fabricantes, ha ralentizado considerablemente la aparición de nuevos productos ZigBee en el mercado.

2.3.1. Consideraciones Generales

A continuación se procederá a describir las características generales más significativas de la especificación ZigBee.

2.3.1.1. Tipos de Dispositivos

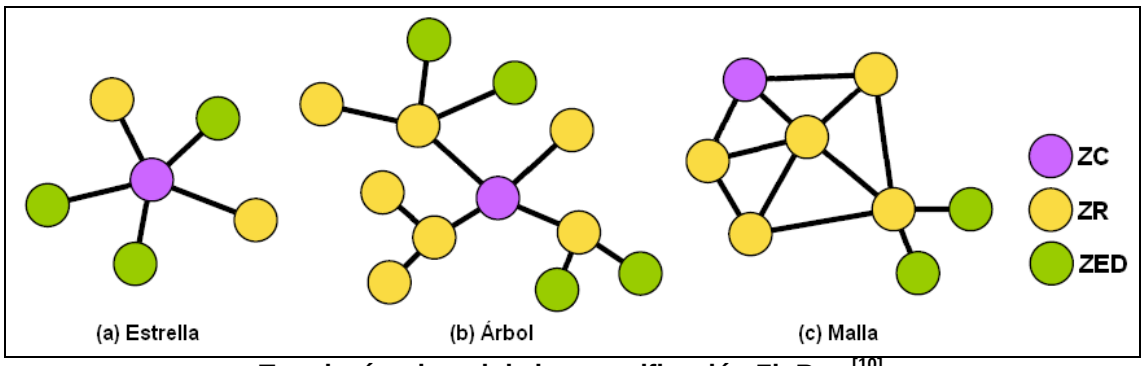
De forma similar a como sucedía en el estándar IEEE 802.15.4 (y basándose en él), ZigBee define diferentes tipos de dispositivo según su papel en la red:

- **Coordinador ZigBee** (*ZigBee Coordinator*, **ZC**). Equivaldrá al coordinador de la PAN según la versión del 2003 del estándar 802.15.4. Siempre deberá existir uno por red ZigBee y será el tipo de dispositivo más completo.
- **Router ZigBee** (*ZigBee Router*, **ZR**). Será un FFD funcionando como coordinador dentro de su espacio de operación (según la definición del estándar 802.15.4-2003), capaz de encaminar mensajes entre dispositivos de la red y de permitir la asociación a la red de nuevos equipos.
- **Dispositivo final** (*ZigBee End Device*, **ZED**). Podrá ser un RFD o un FFD (según la definición del estándar 802.15.4-2003) que no funcione ni como ZC ni como ZR. Poseerá la funcionalidad necesaria para comunicarse con su nodo padre (el coordinador o un router), pero no podrá encaminar información de otros dispositivos. De esta forma, este tipo de nodo puede estar dormido la mayor parte del tiempo, aumentando la vida media de sus baterías. Un ZED suele tener requerimientos reducidos de memoria por lo que habitualmente será significativamente más barato.

De este modo, todas las redes ZigBee dispondrán siempre de un único coordinador ZigBee (ZC) y de un número indeterminado de routers (ZR) y dispositivos finales (ZEDs).

2.3.1.2. Topologías de Red

La capa de red ZigBee (NWK), soportará las topologías de estrella, árbol y malla.



Topologías de red de la especificación ZigBee [10]

En una topología de estrella la red será controlada por un único dispositivo, el coordinador ZigBee (ZC), que será el responsable del control de la asociación y mantenimiento del resto de dispositivos en la red, que se comunicarán directamente con este coordinador.

En las topologías de árbol y malla la red se inicializará y controlará por el coordinador ZigBee pero podrá ser extendida a través de routers (dispositivos FFD funcionando como coordinadores).

En las redes de árbol, los routers (ZC) encaminarán los mensajes de datos o control a través de la red usando una estrategia de enrutado jerárquica y podrán emplear comunicaciones con balizas tal como especifica el estándar 802.15.4-2003 (aunque árbol y estrella serán las únicas topologías que lo permitirán).

Las topologías de malla en cambio permitirán comunicaciones completas punto a punto pero los routers no emitirán balizas periódicamente como define el estándar 802.15.4.-2003.

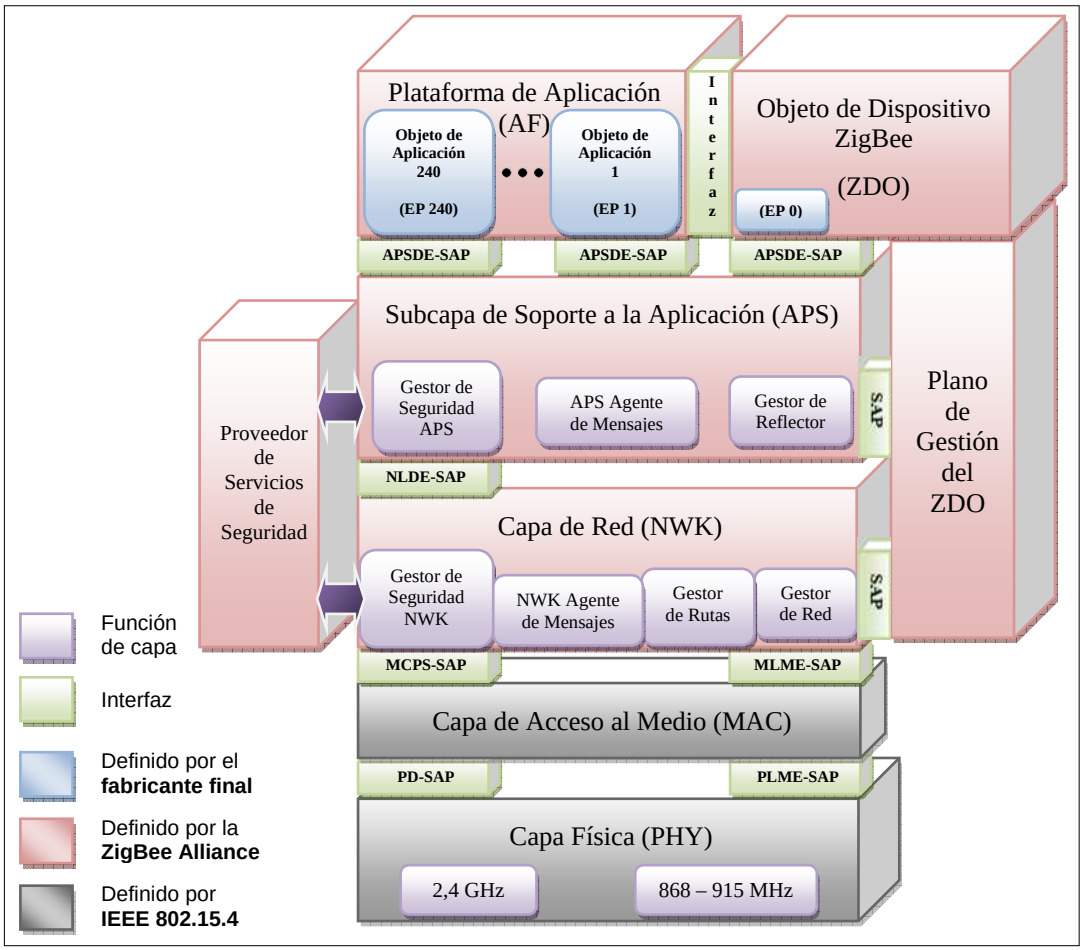
2.3.1.3. Arquitectura de Referencia ZigBee

La arquitectura ZigBee se basa en el modelo de siete capas OSI, dejando las dos capas inferiores, la física y la de acceso al medio (MAC) al estándar 802.15.4 (únicamente a la versión de octubre del 2003).

De este modo, ZigBee especificará únicamente la capa de red, los mecanismos de seguridad a emplear y la interfaz con la capa de aplicación. Esta última estará formada por la subcapa de soporte de la aplicación (APS), el objeto de dispositivo ZigBee ("*ZigBee Device Object*", ZDO) y los objetos de aplicación definidos por el fabricante (que se comunicarán con el resto de la

arquitectura desde la plataforma de aplicación (AF) a través de sus puntos de acceso de servicio).

Así pues, a arquitectura de referencia de la especificación ZigBee se presenta en la siguiente figura:



Esquema de la arquitectura de referencia de ZigBee

La capa de red ZigBee (NWK) incluirá los mecanismos necesarios para desempeñar las funciones de asociación o desasociación de una red, aplicación de seguridad a tramas, encaminamiento de paquetes, descubrimiento y memorización de rutas entre dispositivos, descubrimiento de vecinos adyacentes y almacenamiento de información relativa a éstos entre otras.

La capa NWK de un coordinador ZigBee será pues la responsable de establecer una nueva red cuando sea necesario y de asignar las nuevas direcciones a los dispositivos asociados a ésta.

La capa de aplicación (APL) estará formada por la subcapa de soporte de la aplicación, el objeto de dispositivo ZigBee (ZDO) y la plataforma de la aplicación.

La subcapa de soporte de la aplicación (APS) se encargará principalmente de mantener tablas de “ligaduras” (*bindings*) entre dispositivos con servicios o necesidades similares, convertir direcciones cortas a largas o viceversa, definir direcciones de grupos y procesarlas para el encaminamiento o filtrado de tramas, trocear y reensamblar tramas, y reenviar mensajes entre dispositivos vinculados.

Las responsabilidades del objeto de dispositivo ZigBee (ZDO) consistirán mayormente en la definición del rol que desempeñará el dispositivo en la red (coordinador o dispositivo final, por ejemplo), la gestión de las solicitudes o respuestas de “ligadura”, el establecimiento de sesiones seguras entre dispositivos de la red, y el descubrimiento de dispositivos y los servicios de aplicación que ofrecen.

A continuación se procederá a describir con mayor profundidad las diferentes capas.

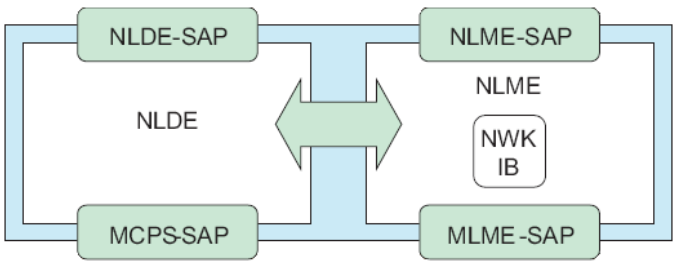
2.3.2. Nivel de Red

La función principal del nivel de red ZigBee debe ser asegurar un correcto uso del nivel MAC especificado en el IEEE 802.15.4-2003 y ofrecer una interfaz adecuada al nivel de aplicación para que éste pueda hacer uso de sus servicios fácilmente.

Este nivel está formado por dos entidades: la entidad de datos NLDE (*Network Layer Data Entity*) y la entidad de gestión NLME (*Network Layer Management Entity*).

La NLDE proporcionará el servicio de transmisión de datos y la NLME se encargará de las operaciones de control y gestión (haciendo uso ocasionalmente de los servicios de la NLDE), para las cuales mantendrá una base de datos conocida como *Network Information Base* (NIB).

Al igual que sucedía con 802.15.4, cada entidad presentará un punto de acceso (SAP) a la capa superior para que ésta pueda hacer uso de los servicios que ofrece. El modelo se presenta en la siguiente figura:



Modelo de referencia de la capa de Red (NWK)

2.3.2.1. Entidad de Datos del Nivel de Red (NLDE)

La NLDE se encargará del transporte de datos de aplicación (APDU) entre dos o más dispositivos pertenecientes a la misma red. De este modo, la NLDE proporcionará los siguientes servicios:

- Generación de tramas de nivel de red (NPDU) a partir de las tramas recibidas del subnivel de soporte de aplicación (ASPDU), mediante la adición de la cabecera apropiada.
- Encaminamiento de mensajes según sea la topología específica que se esté usando.
- Autenticación y confidencialidad de las comunicaciones.

Para cumplir estos objetivos, la NLDE se amparará en las primitivas NLDE-DATA:

Primitiva	Tipo 1	Tipo 2	Tipo 3
NLDE-DATA	Request	Confirm	Indication

Primitivas del servicio de datos capa NWK

Estas tres primitivas **NLDE-DATA** permitirán a la APS solicitar el envío de PDUs destinados a otra u otras APS (.*request*), que los recibirán mediante la primitiva de indicación (.*indication*). El éxito o fracaso del envío se notificará a la APS origen mediante una confirmación (.*confirm*).

2.3.2.2. Entidad de Gestión del Nivel de Red (NLME)

La NLME se encargará de la gestión del nivel de red, permitiendo al nivel de aplicación interaccionar con la pila de protocolos. Para ello, ofrecerá los siguientes servicios:

- Configuración de la pila de protocolos para un funcionamiento correcto.
- Establecimiento de una nueva red ZigBee.
- Asociación, re-asociación y abandono de una red.
- Asignación de direcciones a los dispositivos que se incorporen a la red.
- Descubrimiento, seguimiento y notificación de dispositivos ZigBee adyacentes (vecinos).
- Descubrimiento de rutas para el encaminamiento de paquetes en la red.
- Control del estado activo/inactivo del receptor.
- Encaminamiento de paquetes mediante diferentes mecanismos: Unicast, broadcast, multicast, o many-to-one para el intercambio eficiente de datos en la red.

Una vez más, estos objetivos se alcanzarán mediante el empleo de un número considerable de primitivas que, debido a su número y complejidad, solamente se enumerarán a continuación:

NLME-NETWORK-DISCOVERY, NLME-NETWORK-FORMATION, NLME-PERMIT-JOINING, NLME-START-ROUTER, NLME-ED-SCAN, NLME-JOIN, NLME-DIRECT-JOIN, NLME-LEAVE, NLME-RESET, NLME-SYNC, NLME-SYNC-LOSS, NLME-GET, NLME-SET, NLME-NWK-STATUS y NLME-ROUTE-DISCOVERY.

2.3.2.3. Formato de Trama de Red

El formato de la trama de nivel de red (NPDU) se representa en la siguiente figura:

Octetos: 2	**2**	**2**	**1**	**1**	**0/8**	**0/8**	**0/1**	**Var.**	**Var.**
Control de trama	Dir. destino	Dir. origen	Radio	Nº sec.	Dir. Destino IEEE	Dir. Origen IEEE	Control Multicast	Subtrama de origen de ruta	Datos de trama
Cabecera NWK									Datos NWK

Formato genérico de trama de red (NWK)

Como puede apreciarse en la figura, básicamente se pueden distinguir dos secciones: la cabecera y la carga de datos.

En el campo de control de trama se especificará el tipo de trama (trama de datos o comando NWK), la versión del protocolo empleada, y la activación o desactivación de los mecanismos de: descubrimiento de ruta, envío multicast, seguridad de nivel de red, inclusión de direcciones de 64 bits (largas) origen y/o destino y la inclusión/exclusión de la ruta de origen.

Los campos de dirección de 16 bits incluirán diferentes valores según sea el tipo de envío realizado (Unicast, multicast, broadcast, etc.)

El campo de radio especificará el número máximo de saltos que el paquete puede realizar por la red. Cada dispositivo intermedio que retransmita la trama decrementará en uno su valor.

El resto de campos de la cabecera: direcciones de 64 bits, control multicast y la subtrama de origen de ruta, sólo aparecerán en caso de activarse dichos mecanismos.

Especial mención requiere el campo de subtrama de origen de ruta ya que de activarse, contendrá una lista de los dispositivos intermedios que han ido reencaminando la trama. Este mecanismo permitirá al receptor reencaminar la respuesta a la trama hacia su origen sin requerir de nuevos descubrimientos de ruta, lo cual reducirá las tablas necesarias en memoria de los dispositivos intermedios.

En último lugar, el campo de datos contendrá la trama recibida del nivel de soporte a la aplicación (de tratarse de una trama de datos) o un comando de nivel de red (en el caso de tratarse de un comando NWK).

Los comandos NWK son lo suficientemente complejos, y su relación con el ámbito de este proyecto demasiado distante, como para no profundizar en ellos en esta memoria. Sin embargo se enumeran a continuación: ***Route request***, ***Route reply***, ***Network Status***, ***Leave***, ***Route Record***, ***Rejoin request***, ***Rejoin response***, ***Link Status***, ***Network Report***, y ***Network Update***.

2.3.3. Nivel de Aplicación

El nivel de aplicación (APL) está situado en la parte más alta del modelo de capas definido por la especificación ZigBee y se compone de los siguientes subniveles: el subnivel de soporte de aplicación (APS), la plataforma de aplicación (AF) y el objeto de dispositivo ZigBee (ZDO).

A continuación se procederá a describir resumidamente dichas subcapas y los conceptos relacionados más significativos.

2.3.3.1. Subnivel de Soporte de Aplicación (APS)

La subcapa de soporte de la aplicación (APS) proveerá de un interfaz entre la capa de red (NWK) y la capa de aplicación (APL), a través de un conjunto de servicios que las aplicaciones definidas por los fabricantes compartirán con el ZDO.

De forma análoga a como sucedía en otras capas, en la APS se distingue entre una entidad de datos (APSDE) y una entidad de gestión (APSME), ambos con sus respectivos puntos de acceso (SAP).

Como su nombre indica, la APSDE proveerá del servicio de transmisión de datos entre dos objetos de aplicación de dos dispositivos pertenecientes a la misma red, otorgando además capacidades fiabilidad (mediante reintentos de nivel de aplicación), filtrado de direcciones de grupo, rechazo de mensajes duplicados, fragmentación de paquetes y comunicación a través de “ligaduras” (*bindings*).

Así mismo, la APSME proveerá de diversos servicios a los objetos de aplicación, entre los cuales se incluye la seguridad, la gestión de grupos (para mensajes *multicast*) y los *bindings* entre dispositivos. Adicionalmente se encargará de mantener una base de datos (AIB) de los objetos gestionados.

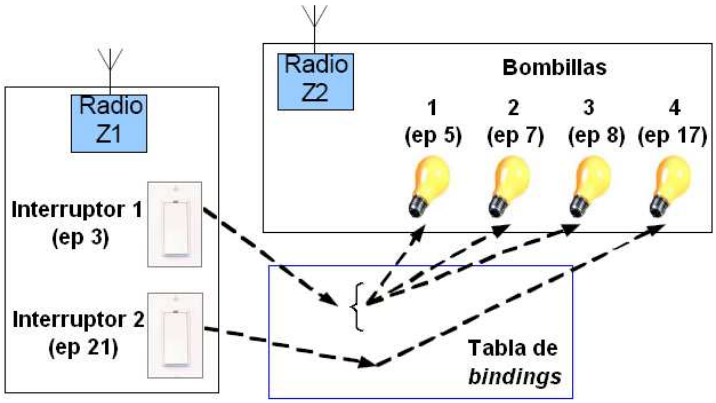
Para alcanzar dichos objetivos, el APS hará uso de la siguiente serie de primitivas: **APSDE-DATA**, **APSME-BIND**, **APSME-UNBIND**, **APSME-GET**, **APSME-SET**, **APSME-ADD-GROUP**, **APSME-REMOVE-GROUP** y **APSME-REMOVE-ALL-GROUPS**.

2.3.3.1.1. Ligaduras (*Bindings*)

Las ligaduras son enlaces virtuales que se establecerán entre *clusters* de dos objetos de aplicación complementarios (ver apartado 2.3.3.2).

Esto quiere decir que será una asociación lógica entre una funcionalidad de una aplicación actuando como servidor y otra funcionalidad compatible de otra aplicación actuando como cliente.

Un ejemplo habitual podría ser el establecimiento de una ligadura entre las actuaciones de encendido/apagado de un interruptor de pared y el encendido/apagado de una bombilla. Este ejemplo se ilustrará en la siguiente figura:



Ejemplo del establecimiento de *bindings* entre dispositivos del perfil *Home Automation*

Como puede observarse en la figura, gracias a los *bindings* un interruptor puede estar ligado a más de una bombilla y controlar su luminosidad simultáneamente. De la misma forma una bombilla podría estar controlada por más de un interruptor.

La información de los *endpoints* y sus *clusters* asociados que están enlazados mediante ligaduras se almacenará en una tabla al efecto que debe conservarse ante posibles reinicializaciones de los dispositivos. Por esta razón dichas tablas se almacenarán habitualmente en el coordinador ZigBee, y sólo se crearán nuevas ligaduras en los escenarios que dispongan de una red ZigBee operativa cuando las restricciones de seguridad aplicadas lo permitan.

Habitualmente los *bindings* se establecerán de forma que sea muy sencillo para el usuario/instalador: mediante la pulsación de un *switch* de los dispositivos implicados, mediante comandos emitidos desde una aplicación controladora, etcétera.

Esto permitirá efectuar la asociación lógica entre dispositivos de funcionalidades similares de forma casi *plug&play*, lo cual otorgará a la red ZigBee de un gran potencial, escalabilidad e interoperabilidad sin necesidad de instalaciones complicadas y costosas.

2.3.3.1.2. Transmisión de Mensajes de Capa de Aplicación

El nivel APS facilitará dos mecanismos para el envío de mensajes entre diferentes *endpoints*: envío directo y envío indirecto.

El **envío directo** asume que se ha realizado un descubrimiento de dispositivo y de servicio previamente y se ha identificado un dispositivo y punto final determinado que puede proporcionar el servicio complementario al que envía los mensajes. Para realizarlo se debe indicar la dirección corta del dispositivo, el *cluster* y el identificador del *endpoint* destino.

Para los dispositivos más simples, la necesidad de incluir toda esta carga de direccionamiento puede ser contraproducente, por lo que suelen emplear el mecanismo de **envío indirecto**, mediante el cual sólo se especifica la dirección del dispositivo y *endpoint* origen y la tabla de ligaduras averiguará el destino o destinos finales. Obviamente la ligadura debe haberse realizado previamente.

Adicionalmente, se permitirá el envío de mensajes ***unicast*** (mediante los mecanismos ya descritos), ***multicast*** (mediante el empleo de direcciones de grupo de 16 bits) y ***broadcast*** (destinados a todos los dispositivos de la red).

Finalmente, como ya se ha mencionado, la capa APS otorgará una comunicación segura y fiable mediante el envío de reintentos y tramas de asentimiento de aplicación (ACK opcionales), además de responsabilizarse de la fragmentación y reensamblado de paquetes cuando sea necesario.

2.3.3.2. Plataforma de Aplicación (AF)

La plataforma de aplicación es el entorno sobre el cual se alojarán los objetos de aplicación ZigBee y que otorgará la interfaz imprescindible para que éstos puedan comunicarse con el resto de las capas que conforman la pila y con los objetos de aplicación de otros dispositivos de la red.

Aunque cada nodo de una red ZigBee estará dotado de una dirección larga de 64 bits y recibirá adicionalmente una dirección corta de 16 bits (identificador de nodo) al asociarse a ésta, se requiere un mecanismo para direccionar cada uno de los distintos objetos de aplicación que pueden residir dentro de un mismo nodo.
De este modo, cada objeto de aplicación se identificará mediante un "*endpoint"* (punto final) numerado con un valor entre 1 y 240. Los valores 0 y 255 identificarán el ZDO y la interfaz de transmisiones *broadcast* respectivamente, mientras que el resto de valores entre 241 y 254 se reservarán para usos futuros.

Adicionalmente, cada objeto de aplicación asociado a un *endpoint* podrá describirse a través de un identificador de tipo de dispositivo que será único dentro de cada **perfil de aplicación** (ver siguiente subapartado) y mediante el conjunto de ***clusters*** y atributos que contiene (ver subapartado 2.3.3.2.2).

2.3.3.2.1. Perfiles de Aplicación (*Profiles*)

Los perfiles de aplicación (*profiles*) son acuerdos de comandos, formatos de mensaje y acciones de procesado, que permitirán a los desarrolladores la creación de una aplicación interoperable distribuida entre las entidades de aplicación residentes dentro de dispositivos de diferentes fabricantes.

Los perfiles de aplicación son una de las características más interesantes del protocolo ZigBee ya que permitirán a diferentes tipos de dispositivos (y a las aplicaciones contenidas en ellos) el intercambio de comandos, solicitudes de datos, y el procesado de los mismos.

Un ejemplo de perfil de aplicación ZigBee podría ser el *Home Automation Profile*, que permitirá a distintos tipos de dispositivos el intercambio de mensajes de control para el establecimiento de una aplicación inalámbrica de automatización del hogar. De este modo, los dispositivos pertenecientes a la red podrán enviar tramas conocidas para apagar/encender luminarias, emitir medidas de temperatura a termostatos, o el envío de alarmas por detección de movimiento, por ejemplo.

Otro ejemplo válido podría ser el perfil de dispositivo ZigBee (ZDP), soportado por todos los dispositivos ZigBee, y que definirá las acciones comunes que tendrán lugar entre dispositivos ZigBee como son las asociaciones a la red y el descubrimiento de dispositivos y servicios.

El primer perfil de aplicación publicado fue el ***Home Automation*** [10], en el año 2007, y define un escenario sencillo, fiable, escalable y de coste moderado para el control de la iluminación, el consumo de energía, el entorno y la seguridad en hogares o pequeñas oficinas.

El segundo perfil publicado fue el ***Smart Energy*** [11], en mayo del año 2008, y ofrece a empresas y a proveedores de energía un sistema sencillo y seguro para la gestión inalámbrica del consumo energético en redes de área del hogar (HAN). De este modo se podrán programar, de forma sencilla, aplicaciones de gestión y eficiencia energética para adecuarse a los requisitos impuestos por el gobierno.

Otros perfiles de aplicación para escenarios tan variados como la automatización de edificios (*Building Automation Profile*) o aplicaciones socio-sanitarias (*Health Care Profile*), se encuentran en proceso de desarrollo y se espera su publicación inminente.

2.3.3.2.2. Racimos (*Clusters*)

Los racimos (*clusters*) son conjuntos de atributos y comandos relacionados que definen un interfaz de comunicación entre dos dispositivos, uno de los cuales empleará las funcionalidades que ofrece como cliente el *cluster* mientras que el otro hará uso de las herramientas que ofrece como servidor.

Los *clusters* se identificarán de forma unívoca dentro mediante un identificador de 16 bits que será único dentro de un determinado perfil de aplicación.

Para los perfiles desarrollados por la *ZigBee Alliance*, ésta ha creado una librería de racimos pública (*ZigBee Cluster Library* [24]) que provee de una definición e identificación común para los *clusters* más comunes (y para los atributos que éstos incluyen). Esta librería se encuentra en constante evolución y mantenimiento, habiéndose publicado ya dos versiones de la misma.

Como ejemplo, se pueden mencionar los *clusters* para la identificación de dispositivos (*Identify cluster*), para la conmutación de estados (*On/Off cluster*), para la medida de temperaturas (*Temperature Measurement cluster*), para la detección de presencia (*Occupancy cluster*) y un largo etcétera.

No obstante, los fabricantes podrán definir sus propios perfiles de aplicación con sus *clusters* y descripciones de dispositivos propietarias (previa negociación con la *ZigBee Alliance*) al igual que podrán incluir determinadas extensiones propietarias dentro perfiles públicos.

Los dispositivos ZigBee, al publicar los servicios que ofrecen, deberán incluir la información relativa al perfil que implementan y a los *clusters* específicos que soportan. Así mismo la creación de ligaduras (*bindings*) entre diferentes *endpoints* se realizará siempre entre dos *clusters* compatibles, uno siguiendo el rol de servidor y el otro el de cliente.

2.3.3.3. Objeto de Dispositivo ZigBee (ZDO)

El objeto de dispositivo ZigBee se comportará como el interfaz existente entre los objetos de aplicación (aplicaciones programadas por el fabricante), el perfil del dispositivo y la plataforma de soporte de la aplicación (APS).

El ZDO satisfacerá los requisitos comunes de todas las aplicaciones que operen empleando la pila de protocolos ZigBee. De este modo, será responsable de:

- Inicializar la capa de soporte de aplicación (APS), la capa de red (NWK) y el proveedor de servicios de seguridad (SSP).

- Recopilar información de configuración de los objetos de aplicación para desempeñar adecuadamente las funciones de descubrimiento de dispositivos y servicios, gestión de seguridad, gestión de red y gestión de *bindings*.

De este modo el ZDO actuará de interfaz público para la plataforma de aplicación (AF), permitiendo a los distintos objetos de aplicación el control del dispositivo físico y el acceso a las funciones de red propias del protocolo.

El objeto de aplicación se localizará siempre en el *endpoint* 0, desde donde se comunicará con las capas inferiores de la pila del protocolo a través de sus puntos de acceso (APSDE-SAP para transmisión de datos y APSME-SAP para mensajes de control).

El servicio de descubrimiento de dispositivos será el proceso mediante el cual un sensor ZigBee podrá conocer otros dispositivos de la red. Para ello permitirá el envío de peticiones de dirección corta o larga de larga de un dispositivo en la red (transmisiones *broadcast* y *unicast* respectivamente) y transmitirá las respuestas al solicitante.

El descubrimiento de servicios será el proceso mediante el cual las capacidades de un determinado dispositivo se publicarán hacia otros dispositivos. Existen varios mecanismos para lograr este objetivo: mediante envíos de solicitudes de información dirigidos a cada *endpoint* de un dispositivo, o mediante el envío del listado de servicios requeridos de un dispositivo y su comparación con los que éste tiene disponibles (*match procedure*).

Así pues se podrá resumir que el ZDO se otorgará las funcionalidades básicas que debe poseer todo coordinador, router o dispositivo final ZigBee para funcionar dentro de la red. De este modo, cada objeto de aplicación se centrará en definir su comportamiento particular.

Existen infinidad de aspectos adicionales de la especificación ZigBee que podrían ser desarrollados, sin embargo, no se profundizará más en ellos debido a su relación débil o nula con el objetivo final de este proyecto. De modo que si se desea adquirir un mayor conocimiento acerca de ZigBee, se recomienda encarecidamente la lectura de la especificación **[6]**.

2.4. Plataformas ZigBee Disponibles en el Mercado

La *ZigBee Alliance* especifica los niveles de la capa de red hasta la capa de soporte de aplicaciones que, combinados con las capas inferiores de IEEE 802.15.4, permiten proporcionar una gama inalámbrica completa.

Fabricantes de semiconductores como *Microchip* [25], *Atmel* [26] y *Freescale* [27] suministran módulos completos de desarrollo que comprenden las capas de ZigBee y IEEE 802.15.4 e integran un microcontrolador para almacenar el *firmware* del protocolo y el de las aplicaciones.

Mediante estos módulos, los ingenieros podrán completar diseños funcionales sin conocimientos especializados en RF y/o en *layout* de placas. A su vez, las placas de desarrollo que soportan estos módulos ofrecerán un punto de partida idóneo para el desarrollo y evaluación de aplicaciones.

En la actualidad, muchos fabricantes de chips ZigBee venden radios integradas y microcontroladores dentro de un mismo chip (*single chip*) con una capacidad de entre 60 KB a 128 KB de memoria Flash. Tal es el caso del JN5148 (*Jennic*), el MC13213 (*Freescale*), el EM250 (*Ember*) o el CC2430 (*Texas Instruments*). Habitualmente la radio también puede adquirirse de forma separada, permitiendo a los desarrolladores elegir el microprocesador que deseen.

De forma adicional, la mayoría de fabricantes de chips ZigBee suministrarán la pila de protocolos (802.15.4 y ZigBee) e incluso las fuentes de su código para los chips que venden.

Una selección adecuada de un determinado kit de desarrollo ZigBee estará determinada por la profunda evaluación de los siguientes factores:

1. Costes de las licencias del entorno de desarrollo y de la pila de protocolos.
2. Coste y eficiencia del servicio técnico (soporte).
3. Coste de los analizadores de protocolo (*sniffers*), de ser necesarios.
4. Completitud de la pila de protocolos suministrada (cosa poco habitual).
5. Inclusión (o no) de perfiles públicos de aplicación en la pila (cosa aún menos habitual).
6. Usabilidad del IDE y del depurador.
7. Inclusión de ejemplos de aplicación (que acelerarán el proceso de desarrollo).
8. Inclusión de herramientas gráficas para la configuración y generación de aplicaciones de ejemplo personalizadas (para un hardware y comportamiento específicos).
9. Inclusión de un mecanismo de carga remota (junto con un bootloader) para la actualización inalámbrica del *firmware* de los equipos.

10. Disponibilidad de una documentación completa de las librerías, ejemplos de aplicación y entornos de desarrollo.

De este modo, existe en la actualidad un número considerable de empresas que suministran chips ZigBee, OEMs, kits de desarrollo, módulos de radio, o soluciones de integración de los chips de otros fabricantes. Muchos de estos productos se encuentran ya certificados por la *ZigBee Alliance*, pero no todos.

En la siguiente tabla se ilustran las principales características de los productos ofrecidos por los fabricantes de chips más importantes (que además permanecen activos desde antes del inicio de este proyecto):

Compañía	Productos	Notas
Freescale [14]	MC13211 (8 bits) MC13212 (8 bits) MC13213 (8 bits) MC13224V (32 bits, ARM7)	Parte de Motorola hasta el 2004. Suministra chips ZigBee (soluciones *single chip* o *dual chip*), transceptores radio, kits de desarrollo, pilas de protocolos (*BeeStack*), aplicaciones de ejemplo y herramientas SW.
Ember [15]	EM250 (8 bits) EM260 (8 bits) EM351 y EM357 (32 bits, ARM)	Una de las empresas fundadoras de la ZigBee Alliance. Suministra chips ZigBee (*SoC* o coprocesador separado), kits de desarrollo, pilas de protocolo (*EmberZNet*), aplicaciones de ejemplo, analizador de protocolos y herramientas SW.
Texas Instruments [16]	CC2430/1 (8 bits) CC2530 (8 bits) CC2480 (8 bits) MSP430X	Dueña de Chipcon y promotora de la ZigBee Alliance. Suministra chips ZigBee (soluciones *single chip*, *dual chip o coprocesador separado*), transceptores radio, kits de desarrollo, pilas de protocolos (*ZStack*), aplicaciones de ejemplo y herramientas SW.
Jennic [28]	JN5148 (32 bits) JN5139 (32 bits) JN5121 (32 bits)	Empresa participante de la ZigBee Alliance. Suministra chips ZigBee (soluciones *single chip o coprocesador separado*), kits de desarrollo, pilas de protocolos y herramientas SW.

Principales fabricantes de chips ZigBee

Basándose en los chips mencionados, se encuentran en la actualidad multitud de empresas que suministran soluciones ZigBee de todo tipo (completas o parciales).

Algunas de las más importantes serán: **CEST** (con soluciones basadas en chips de *Texas Instruments*), **MeshNetics** (basadas en los transceptores radio de *Atmel, ZigBits*), **MaxStream/Digi** (basados originalmente en *Freescale* y ahora en *Ember*), **Telegesis** (basados en *Ember*) o **Crossbow** (basados en *Atmel*).

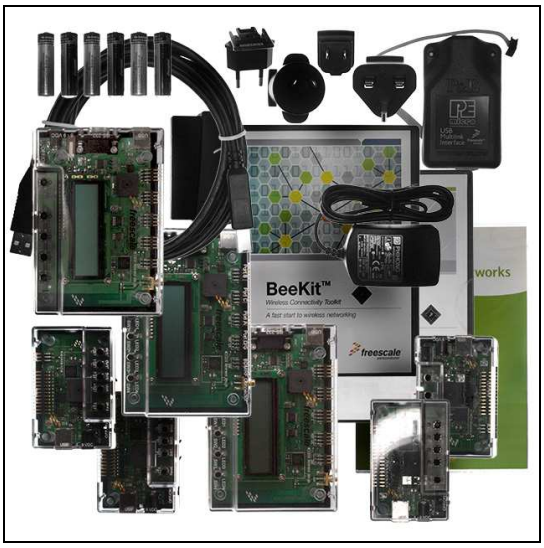
En el caso de España, una de las soluciones más atractivas, por la variedad de productos disponibles y por el tiempo que permanece en el sector, es la empresa **NLaza Soluciones** [19].

NLaza, en su línea de productos *NDimension*, suministra dispositivos 802.15.4 y ZigBee basados en el chip MC13213 de *Freescale* para una gran variedad de escenarios como son la domótica, las aplicaciones socio-sanitarias, la seguridad, el control de consumo energético, etc.

Aunque en la actualidad resulta sencillo el acceso y adquisición de los kits de desarrollo y productos finales 802.15.4 y ZigBee mencionados, esto no sucedía así hace unos años (cuando se inició este proyecto), ya que la tecnología aún era inmadura y no tenía de tantos seguidores.

En las fechas en que se eligió el fabricante las únicas referencias sólidas eran *Freescale* (anteriormente parte de *Motorola* [29]) y *Texas Instruments*. Existía una tercera compañía, *Ember*, que aún siendo una de las fundadoras de la *ZigBee Alliance*, no se tenía claro si tendría la capacidad de superar la crisis del sector (aunque luego se ha convertido en uno de los referentes del sector).

Freescale suministraba entonces un kit de desarrollo muy completo para el chip MC13213 (mucho más que el de la competencia), satisfaciendo adecuadamente los 10 requisitos enumerados anteriormente. Simultáneamente, *NLaza Soluciones* era una de las pocas empresas españolas que fabricaban sus dispositivos ZigBee basándose en ella.



Kit de Desarrollo MC13213 de *Freescale*

De este modo, la elección de la plataforma ZigBee de *Freescale* (Kit MC13213) y los dispositivos de *NLaza Soluciones* (Interfaz RS232 ND07) como base para la realización de este proyecto fueron claramente la decisión a tomar en ese momento.

2.5. Soluciones de Autoconfiguración de Dispositivos

Como ya se ha mencionado al principio de esta memoria, y se desarrollará con profundidad a lo largo de los siguientes capítulos, las redes ZigBee requieren de un mecanismo que permita la actualización/configuración remota de los dispositivos que la integran.

Este mecanismo estará formado por un protocolo de carga remota y un cargador de arranque (bootloader) capaz de actualizar el *firmware* del dispositivo.

Puesto que no hay una especificación regulatoria al respecto, el protocolo empleado puede ser cualquiera que otorgue las suficientes garantías de velocidad, fiabilidad y seguridad. Soluciones habituales del mercado suelen basarse en protocolos inalámbricos como *TFTP* o *XModem-CRC*. Aún así este campo es todavía objeto de estudio en la actualidad.

Sin embargo, el cargador de arranque es absolutamente dependiente del hardware concreto que está destinado a actualizar, de modo que su funcionamiento será específico a cada tipo de dispositivo ZigBee.

Es por ello que, si se desea desarrollar un bootloader muy potente, capaz de detectar múltiples situaciones de error y protegerse ante ellas (como disponer de un estado radio de recuperación del dispositivo) será necesario disponer de mucha memoria y de acceso completo al código fuente de las librerías ZigBee, situación harto improbable (salvo para el fabricante).

Recientemente, Ember ha incluido dos tipos de bootloader para los chips que suministra (EM250 y EM260 principalmente), el "*application bootloader*" y el "*standalone bootloader*".

El "*application bootloader*" es un pequeño trozo de código que se compila junto con la aplicación principal permitiendo la actualización remota de dispositivos siempre y cuando dispongan de una memoria auxiliar integrada (de unas marcas concretas) para el almacenamiento de los nuevos *firmwares*. Este bootloader ocupará poca memoria Flash pero dependerá completamente de la aplicación principal para la obtención de las nuevas imágenes de *firmware* mediante la recepción de paquetes de nivel de aplicación (que recibirá mientras se encuentra activo en la red ZigBee).

Por el contrario, el "*standalone bootloader*" es una aplicación completa (ocupa mucha más memoria) que se suministra precompilada y que se almacena en el chip de forma separada al programa principal. Esta aplicación permitirá la actualización del *firmware* mediante comandos de bajo nivel (capa de enlace de 802.15.4) a través de un protocolo simple basado en *XModem-CRC*. Proveerá de un mecanismo de recuperación (radio y serie) ante situaciones de corrupción del *firmware* pero, en cambio, no permitirá (al no emplear una memoria auxiliar) el

funcionamiento en red del dispositivo mientras está siendo actualizado.

Estos dos ejemplos ilustran la problemática del escenario: a mayor complejidad del protocolo o protección frente a errores, será necesario el uso de más memoria y acceso a las librerías internas del protocolo. Adicionalmente, en todos los casos, el funcionamiento de la solución de actualización será completamente dependiente del hardware a actualizar.

Freescale en cambio, no dispone a priori de un mecanismo inalámbrico de actualización de dispositivos, pero anuncia la disponibilidad de un bootloader serie [30] [31], capaz únicamente de transferir imágenes a un receptor a través de una comunicación RS232. Este "bootloader" no es un mecanismo de actualización remota de dispositivos, y se suministra precompilado y hermético.

Es en este escenario cuando el bootloader objetivo de este proyecto cobra radical importancia, ya que viene a cubrir una funcionalidad esencial de la tecnología que faltaba por incorporar.

De este modo, en el escenario concreto que representan el chip MC13213, un acceso limitado a los códigos fuentes de las librerías de *Freescale*, y los periféricos y la memoria auxiliar incluidos en el ND07, se ha desarrollado un bootloader robusto y eficiente, independiente y compatible con cualquier protocolo inalámbrico (o cableado) que se emplee para recuperar las imágenes del nuevo *firmware*.

Así pues, el bootloader desarrollado en este proyecto permitirá a desarrolladores de aplicaciones disponer de un mecanismo de carga remota de dispositivos que serán capaces de seguir funcionando en red mientras reciben la nueva imagen de *firmware* a través del protocolo programado (TFTP, X-Modem, propietario, etc.) y dentro del tipo de red seleccionado (802.15.4 o ZigBee).

En los siguientes capítulos se desarrollará en profundidad el funcionamiento del bootloader desarrollado y los mecanismos empleados para su depuración y validación.

Capítulo 3
Escenario de Aplicación y Requisitos

3.1. Introducción

En este capítulo se procederá a describir en primer lugar los requisitos que una red de sensores inalámbricos de bajo consumo impone a la hora de elegir una plataforma hardware concreta.

Posteriormente se plantearán los requisitos que han de satisfacerse para lograr que una aplicación de actualización remota de dispositivos, y en concreto la parte relativa al bootloader, pueda ser instalada con éxito en el hardware elegido.

A continuación se introducirá el escenario de aplicación formado principalmente el dispositivo ZigBee, el microcontrolador que integra, las librerías de *Freescale* y el entorno de desarrollo que se han seleccionado con el fin de cumplir con los requisitos anteriormente expuestos.

En último lugar se introducirán dos protocolos de comunicación, el bus I^2C y el estándar RS232, cuyo estudio será necesario para la correcta programación de las aplicaciones que se pretenden desarrollar en este proyecto.

3.2. Requisitos de una Red de Sensores de Bajo Consumo

Las redes de sensores de bajo consumo representan una subclase especial dentro de las redes de sensores inalámbricos, debido a sus interesantes propiedades y a las limitaciones que imponen.

Como su nombre indica, una de las prioridades de este tipo de redes será minimizar al máximo el consumo de cada dispositivo de forma que su instalación represente un ahorro considerable frente a otras soluciones posibles.

Gracias a esta propiedad de bajo consumo, será posible que la mayor parte de los dispositivos integrantes de la red sean alimentados mediante baterías en lugar de a través de la red eléctrica (ya que la duración de éstas será lo suficientemente prolongada como para que compense el esfuerzo de cambiarlas o recargarlas periódicamente).

Esto no sólo representará una ventaja monetaria, sino que también otorgará de mayor flexibilidad y versatilidad al dispositivo, ya que su instalación será mucho más sencilla y no precisará, en la mayoría de los casos, de realizar una obra complicada para extender la alimentación hasta el nuevo emplazamiento del sensor.

Otra propiedad buscada habitualmente en los dispositivos de las redes de sensores de bajo

consumo (dependiendo del mercado al que esté dirigido el dispositivo) es que sea de tamaño reducido y de funcionamiento sencillo, simplificando al máximo el procedimiento de configuración y el impacto visual que produce en el entorno (algo realmente importante en entornos domóticos, por ejemplo).

Se espera que las redes de sensores sean escalables, de forma que un sistema pueda ir incorporando nuevas funcionalidades y mejorando el servicio mediante la inclusión de nuevos dispositivos. Para que esto sea posible, la red deberá ser capaz de soportar el funcionamiento de un número muy elevado de dispositivos de forma simultánea y el coste individual de éstos deberá ser reducido.

Adicionalmente, se espera que estas redes sean capaces de proporcionar un sistema seguro y robusto, capaz de asegurar unas comunicaciones fiables y una fuerte protección frente a interferencias. Para alcanzar este objetivo será necesario seleccionar un protocolo capaz de cumplir con dichos requisitos y que precise de una cantidad reducida o moderada de recursos para lograrlo. La especificación ZigBee ha sido específicamente diseñada para ese fin.

De este modo las redes de sensores inalámbricos de bajo consumo dispondrán de una considerable ventaja competitiva frente a otras redes cableadas puesto que el coste unitario de cada dispositivo (al disponer de un hardware muy sencillo su precio debería de ser bajo) y el de instalación serán considerablemente más asequibles.

Es por todo esto que las redes de sensores de bajo consumo son atractivas para un conjunto tan variado de sectores como son: la domótica y automatización del hogar, entornos socio sanitarios, seguridad de edificios e infraestructuras, monitorización del entorno, sensorización industrial, control de tráfico, aplicaciones militares, y un largo etcétera.

No obstante, no todo son ventajas y la reducción de consumo y tamaño de los dispositivos de la red se traducirá habitualmente en un aumento de la latencia de las comunicaciones, una reducción del ancho de banda y un empeoramiento de la calidad de servicio.

El reducido tamaño de los dispositivos puede llegar a ser una desventaja para muchas aplicaciones industriales, donde el factor estético y la alimentación mediante baterías no son características determinantes.

Por otro lado, la consecución del bajo consumo suele implicar el desarrollo de aplicaciones software más complicadas, capaces de encaminar paquetes dentro de sistemas seguros de alta latencia, debida a que un gran número de dispositivos apagará su interfaz radio la mayor parte del tiempo para ahorrar baterías.

Se puede concluir pues, que la elección de un dispositivo adecuado para funcionar dentro de una red de sensores de bajo consumo suele implicar que disponga de las siguientes características:

- Soporte de un protocolo de comunicaciones inalámbricas robusto, fiable, seguro y relativamente sencillo, para poder programarlo sobre dispositivos de memoria reducida
- Presente un consumo reducido o disponga de mecanismos para reducirlo la mayor parte del tiempo que dure su vida útil
- Tamaño reducido
- Largo o medio alcance de sus transmisiones de paquetes
- Sencillez de instalación y configuración
- Bajo coste unitario
- Fácil de adquirir (muchos dispositivos en fases de desarrollo o productos del mercado son difíciles de conseguir sin acuerdos comerciales o un pedido de gran magnitud de por medio)

Como se explicará más adelante, el dispositivo ND07 de *NLaza Soluciones* y la plataforma que integra (MC13213 de *Freescale*) cumplen todos los requisitos expuestos y representarán por ello una base óptima sobre la que desarrollar este proyecto.

3.3. Requisitos de una Aplicación de Actualización de *Firmware*

Una vez se dispone de una red de sensores de bajo consumo surge la necesidad de desarrollar un sistema para su adecuado mantenimiento y actualización. Estas actualizaciones serán imprescindibles para la corrección de errores y la incorporación de nuevas funcionalidades.

Desafortunadamente, este proceso de actualización puede llegar a ser muy complicado, ya que dichas redes pueden llegar a estar compuestas por centenares o miles de dispositivos, repartidos en todo tipo de áreas. Si además se tiene en cuenta la alta latencia de determinadas comunicaciones, la tarea de identificar los dispositivos a actualizar podría llegar a ser un problema de difícil solución.

A todo esto han de sumarse aquellos escenarios en los que los dispositivos están emplazados en localizaciones de difícil acceso, ocultos del público (intencionadamente o no), protegidos de agentes externos dentro de cajas de registro, etcétera.

Por otro lado, debido a la gran diversidad de sensores disponibles en el mercado, no es posible garantizar la disponibilidad de un mecanismo de actualización manual común a todos los

dispositivos que, con el fin de reducir costes, minimizarán la integración de periféricos al máximo. De este modo es imposible contar con que la mayoría de sensores dispondrá de un puerto serie RS232, una conexión USB o similar. El único mecanismo siempre disponible para su configuración, que sea común a todos ellos, será la comunicación inalámbrica.

De este modo, parece ponerse de manifiesto que la actualización manual de los dispositivos será desaconsejable en casi todos los escenarios posibles. Todo apunta a que la solución óptima (de existir una) será siempre una solución inalámbrica.

Por otro lado, toda aplicación de actualización de software se puede separar en dos partes: el "protocolo" de carga remota (encargado de obtener el nuevo *firmware*) y el bootloader (encargado de actualizar la memoria del dispositivo y reiniciarlo).

Puesto que la carga remota básicamente será una funcionalidad añadida sobre la aplicación principal del dispositivo (independiente del hardware sobre el trabaje), su instalación requerirá esencialmente del cumplimiento de ciertos requisitos de espacio libre en memoria y coexistencia con el resto del software del dispositivo. Ninguno de estos requisitos será necesariamente sencillo de alcanzar puesto que:

- La memoria disponible en dispositivos de bajo consumo suele ser un recurso escaso.
- La coexistencia dentro de un mismo dispositivo entre una aplicación que requiere la transferencia de un elevado volumen de datos (carga remota) y una aplicación principal de bajo consumo (que suele apagar la radio para reducir el consumo) suele ser un proceso delicado.

Sin embargo el bootloader, objetivo de este proyecto, al tener una fuerte dependencia con el hardware sobre el que se instale, impondrá los mismos requisitos que el protocolo de carga remota más los adicionales relativos al dispositivo físico a actualizar. Así pues, los requisitos que una determinada plataforma ha de satisfacer para poder instalar adecuadamente un bootloader sobre ella serán los siguientes:

- Disponer de una memoria auxiliar con capacidad suficiente como para almacenar una imagen de *firmware* (a ser posible capacidad para la imagen más grande que se vaya a generar).
- Disponer de un mecanismo de grabación/recuperación de dicha imagen sobre/desde la memoria auxiliar (usualmente mediante un protocolo de comunicación entre circuitos integrados: UARTs, I^2C, SPI, etc.).
- Disponer de suficiente memoria RAM como para poder leer los datos desde la memoria auxiliar y procesarlos para actualizar el *firmware* del dispositivo.

- Disponer de suficiente memoria no volátil (ROM o Flash) para almacenar el código del bootloader junto con el de la aplicación principal.
- Disponer de un mecanismo de borrado y sobrescritura de la memoria no volátil (ROM o Flash) en tiempo real de ejecución.
- Disponer de un mecanismo software que permita resetear el dispositivo, para así reinicializar automáticamente su estado una vez se ha finalizado la actualización.

De forma equivalente, las características que presentan las redes de sensores de bajo consumo así como el hardware y el software de los equipos que las conforman, determinarán en gran medida el diseño final del bootloader. Así pues, el desarrollo adecuado de la aplicación del bootloader sobre un modelo de dispositivo concreto dependerá de:

- Las librerías del protocolo inalámbrico instalado (ZigBee y 802.15.4, a efectos de este proyecto). La documentación disponible sobre ellas y la posibilidad de modificarlas o acceder a sus fuentes.
- La secuencia de arranque de los diversos componentes del chip y la posibilidad de emplazar el bootloader dentro de ella.
- El funcionamiento de los diversos módulos que integra el dispositivo y el funcionamiento de las librerías que los controlan.
- La posibilidad de funcionar de forma transparente al resto de las aplicaciones programadas en el dispositivo (coexistiendo con ellas sin afectarlas negativamente). Esto dependerá principalmente del acceso que se disponga al “sistema operativo” del microprocesador integrado.

Adicionalmente, dada la naturaleza del escenario de aplicación, el bootloader deberá perseguir siempre los siguientes objetivos:

- Minimizar el consumo de memoria (RAM y ROM) y de capacidad de proceso en el desempeño de todas sus funciones.
- Minimizar su impacto sobre el resto de aplicaciones del dispositivo. De esta forma el bootloader será más escalable y flexible (pudiéndose modificar el mecanismo de carga remota en un momento dado, sin alterar el bootloader).

Dado el escenario de aplicación descrito y todos los requisitos impuestos por los componentes involucrados en el sistema, es necesario poner énfasis en que el número de soluciones posibles dependerá exclusivamente de los parámetros anteriormente enumerados.

A lo largo de este proyecto se demostrará que, empleando ZigBee como protocolo inalámbrico de bajo consumo y los dispositivos ND07 como soporte hardware de la aplicación, es posible desarrollar una solución robusta de bootloader que cumpla con los requisitos expuestos.

3.4. Dispositivo NLaza NDimension ND07

El módulo ND07, de la familia de productos domóticos NLaza NDimension, es un dispositivo diseñado para funcionar como coordinador de red ZigBee capaz de inicializar una red inalámbrica y de gestionarla mediante los comandos recibidos a través de una comunicación serie RS232.

La aplicación más habitualmente programada en el ND07 permitirá configurar los dispositivos pertenecientes a la red que ha generado y se encargará de almacenar los datos emitidos por éstos (principalmente sensorizaciones) para después comunicarlos al exterior vía RS232 hacia una aplicación compatible.

De este modo, cuando el ND07 se programe para integrarse dentro de una red ZigBee que cumpla el perfil de automatización del hogar (*HA*), lo más habitual será que actúe como un dispositivo de Interfaz Combinado (*HA Combined Interface*). Esto quiere decir que funcionará principalmente como una pasarela entre la red ZigBee y otra red o aplicación externa, facilitando la comunicación entre ambas mediante una comunicación serie RS232.

Así pues, las principales características del dispositivo son las siguientes:

- Incorpora la plataforma MC13213 de *Freescale* integrada en la PCB.
- Dispone de una memoria auxiliar externa EEPROM AT24C512 de 64KB.
- Puerto Serie RS232 con conector DE-9 hembra.
- Antena fractal omnidireccional capaz de emitir en la banda de frecuencias sin licencia de 2,4GHz.
- Incluye un pulsador para resetear el dispositivo y otro configurable.
- Integra un indicador LED para notificar el estado de la aplicación.
- Presenta un consumo muy reducido.
- Alimentación mediante transformador a 4,5v.
- Dispositivo pequeño, portable e intuitivo.

Dispositivo NLaza NDimension ND07 (con caja)

A continuación se procederá a describir con mayor detalle sus especificaciones.

3.4.1. Especificaciones Eléctricas y Mecánicas [32]

El ND07 es un dispositivo de consumo reducido diseñado para funcionar como coordinador de una red ZigBee, de forma que requerirá de un suministro de corriente constante proveniente de la red eléctrica.

De este modo, sus principales especificaciones se resumirán en la siguiente tabla:

Parámetro	Condición	Valor Mínimo	Valor Típico	Valor Máximo	Unidades
Tensión de Alimentación		3.3	4.5 [I]	6.5	V
Tensión de Alimentación (por pines de la placa)		2.5	2.7 [II]	2.7	V
Frecuencia de Portadora		2405	2475 [III]	2480	MHz
Alcance	[IV]	25	30	35	m
Consumo Medio			60[V]		mA
Velocidad de Transmisión	[VI]		4800		baudios
RS232: Bits de Datos			8		bits
RS233: Bits de Paridad			0		bits
RS233: Bits de Parada			1		bits
RS233: Control de Flujo	[VII]		Hardware		-
Dimensiones			71x66x28		mm

[I]: Tensión continua de salida del adaptador.

[II]: Este modo de alimentación carece de circuito de protección y puede dañarse el dispositivo permanentemente. No se recomienda emplearlo salvo por desarrolladores expertos mientras dure el proceso de programación y depuración.

[III]: Frecuencia del canal 25 de 802.15.4/ ZigBee. Puede llevar cualquiera de los 16 canales posibles configurado de fábrica.

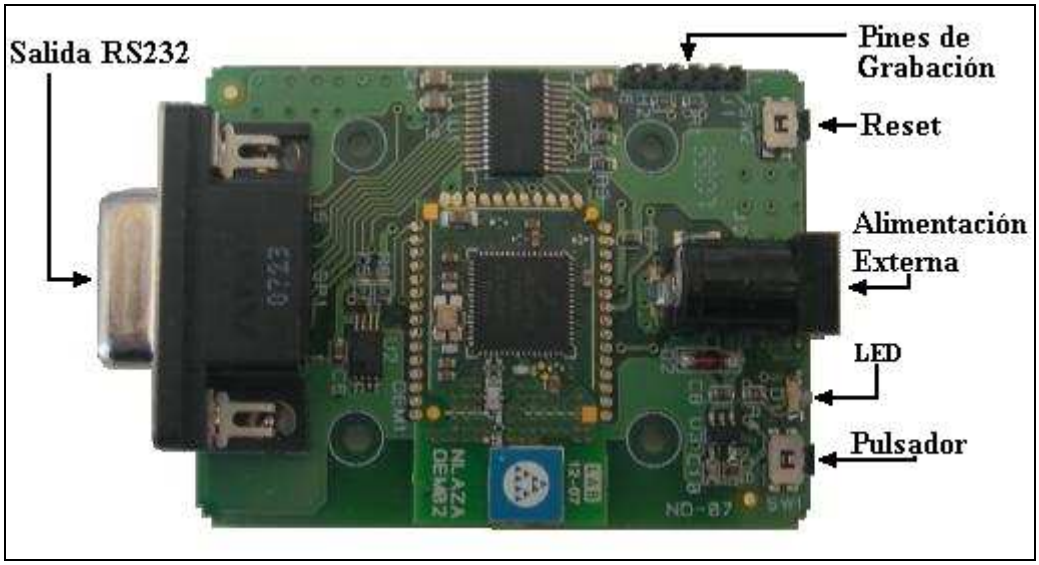
[IV]: Alcance medido en condiciones de espacio libre y en canal 25.

[V]: Consumo medio medido en situación estable, con el dispositivo asociado a la red ZigBee.

[VI]: Aunque el microcontrolador MC13213 junto con el chip MAX3232 integrado soportan velocidades serie superiores, ésta es la velocidad media recomendada para una óptima coexistencia con la aplicación ZigBee.

[VII]: Es imprescindible habilitar el control de flujo Hardware en la aplicación externa que recibe las tramas enviadas por el ND07 para asegurar un correcto funcionamiento.

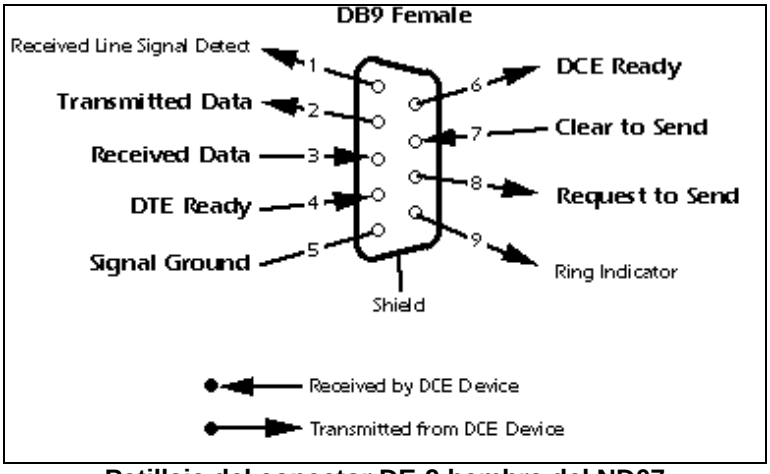
Como puede observarse en la siguiente imagen, aunque el ND07 es un dispositivo de dimensiones reducidas, ha sido diseñador para ser muy intuitivo y cómodo a la hora de desarrollar aplicaciones para él:



Placa ND07 con el chip MC13213

Cuando el ND07 se presenta con su encapsulado habitual, el LED luminoso y los pulsadores serán accesibles mediante unos pequeños orificios situados en la parte posterior de la caja.

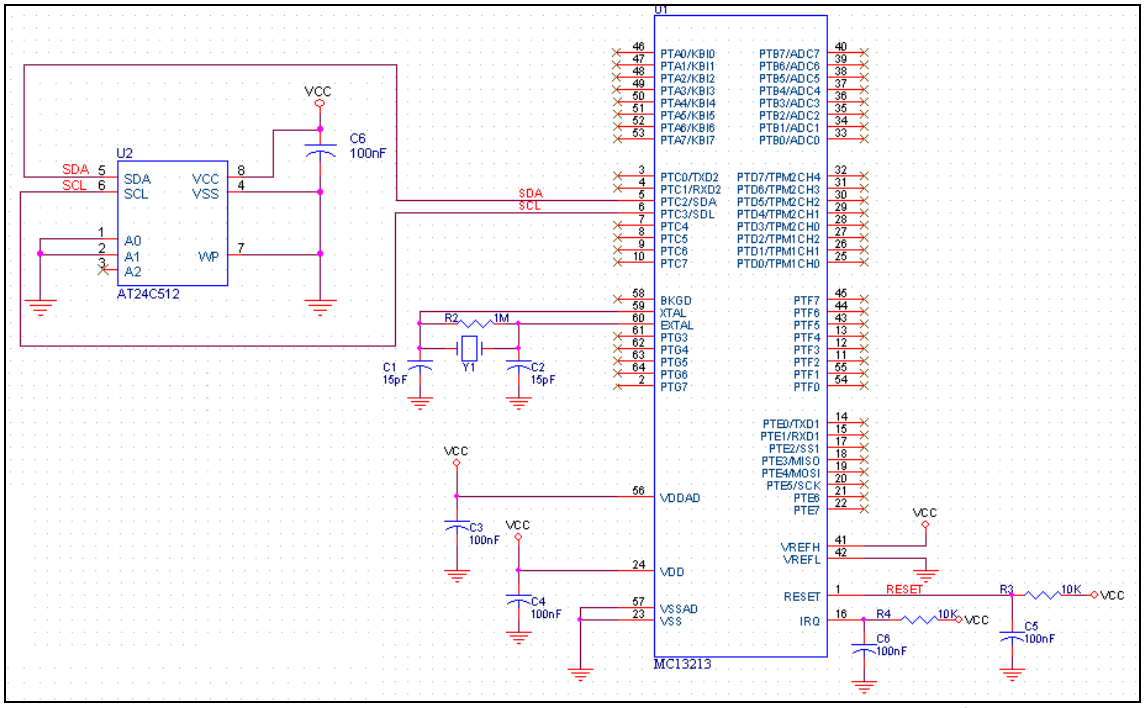
El ND07 dispone de un conector serie DE-9 hembra para conectarse al puerto serie de un ordenador (a través de una comunicación RS-232) mediante un cable apropiado. El patillaje será el siguiente:



Patillaje del conector DE-9 hembra del ND07

El dispositivo se conectará a la red eléctrica mediante un adaptador a 4.5 voltios en escenarios normales de operación. Sin embargo, mientras dure el proceso de desarrollo y depuración, será necesario alimentarlo a través de los pines de la placa mediante una fuente de alimentación regulada a 2,7 voltios.

Adicionalmente, el ND07 dispone integrado en placa de una memoria EEPROM de 64KB (AT24C512 de *Atmel*) que permitirá el almacenamiento dinámico de datos no volátiles en tiempo de ejecución. Este chip se comunicará con la plataforma MC13213 mediante un bus I^2C, y se conectarán tal y como se presenta en la siguiente figura:



Esquema de Conexionado MC13213 y AT24C512 mediante bus I^2C

Como puede observarse del esquema anterior, el conexionado de los pines A0, A1 y A2 de la memoria EEPROM AT24C512 fijará la dirección de este integrado dentro del bus I^2C.

La hoja de especificaciones de la EEPROM indica que el pin A2 debe dejarse al aire, sin conectar, y que serán los pines A0 y A1 los únicos que influirán en la dirección I^2C final del dispositivo.

Puesto que ambos pines están conectados a tierra, los dos bytes menos significativos de la dirección del AT24C512 serán cero. Los 5 bytes más significativos restantes los fuerza el propio dispositivo al valor 10100 (en binario).

De este modo la EEPROM empleada en el ND07 dispone de la dirección binaria I^2C **1010000** (7 bits en binario) dentro del bus. Este dato será fundamental más adelante en el desarrollo del proyecto para direccionar correctamente el integrado.

3.4.2. Plataforma MC13213 *System-On-Chip* de *Freescale*

La característica más destacable de un dispositivo ZigBee será sin duda alguna el microcontrolador que integra, ya que será éste el que definirá los protocolos inalámbricos que se podrán instalar y los periféricos que se podrán controlar.

De este modo, a efectos de este proyecto, será imprescindible dedicar tiempo al estudio de la plataforma MC13213 (ya los módulos que incluye), ya que será la base del funcionamiento del ND07 de NLaza Soluciones.

La solución MC13213 de *Freescale* es una plataforma ZigBee de segunda generación que incorpora un transceptor radio de baja potencia en la banda de frecuencias de 2,4 GHz y un microcontrolador de 8 bits todo dentro de un encapsulado de 71 pines y dimensiones de 9x9x1 mm. La combinación de estos dos componentes dentro del mismo chip de reducidas dimensiones permite obtener un producto muy versátil, sencillo y de coste reducido.

Esta plataforma puede ser empleada para desarrollar productos que van desde simples aplicaciones propietarias que permitan conectividad punto a punto hasta soluciones completas ZigBee de red mallada conformes a la especificación.

Así pues, las aplicaciones más habituales para las que se suele emplear el MC13213 incluyen escenarios de automatización residencial o comercial, control industrial, socio sanitarios y bienes de consumo.

Este chip ha sido específicamente diseñado con la intención de albergar aplicaciones basadas en los protocolos 802.15.4 y ZigBee mediante las librerías 100% compatibles (*MAC* y *BeeStack*) suministradas por *Freescale*. Adicionalmente provee soporte para otras librerías de SMAC (Simple MAC), *SynkroRF* y ZigBee *RF4CE*.

Una de sus características más destacables es que contiene un transceptor radio capaz de operar en la banda de frecuencias sin licencia de 2,4GHz, en dieciséis canales distintos y con una potencia de salida regulable.

Este transceptor radio incluye un amplificador de bajo ruido, un *switch* integrado de transmisión/recepción, regulación de la alimentación de la placa, y la capacidad de codificar y decodificar señales de espectro ensanchado. La potencia nominal de salida será de 1mW.

La plataforma también contiene un microcontrolador basado en la familia de MCUs HCS08 que dispone de 60KB de memoria Flash y 4KB de memoria RAM. Este controlador tendrá la

suficiente capacidad como para que la pila de comunicaciones y la aplicación desarrollada residan ambas dentro del mismo encapsulado (*System in Package*, SiP).

De este modo, en la arquitectura del MC13213 se pueden distinguir dos bloques claramente diferenciados, el módem radio (802.15.4) y el microcontrolador de 8 bits 9S08GB60. Dicha distinción de bloques y los módulos incluidos en cada uno pueden apreciarse en la siguiente figura:

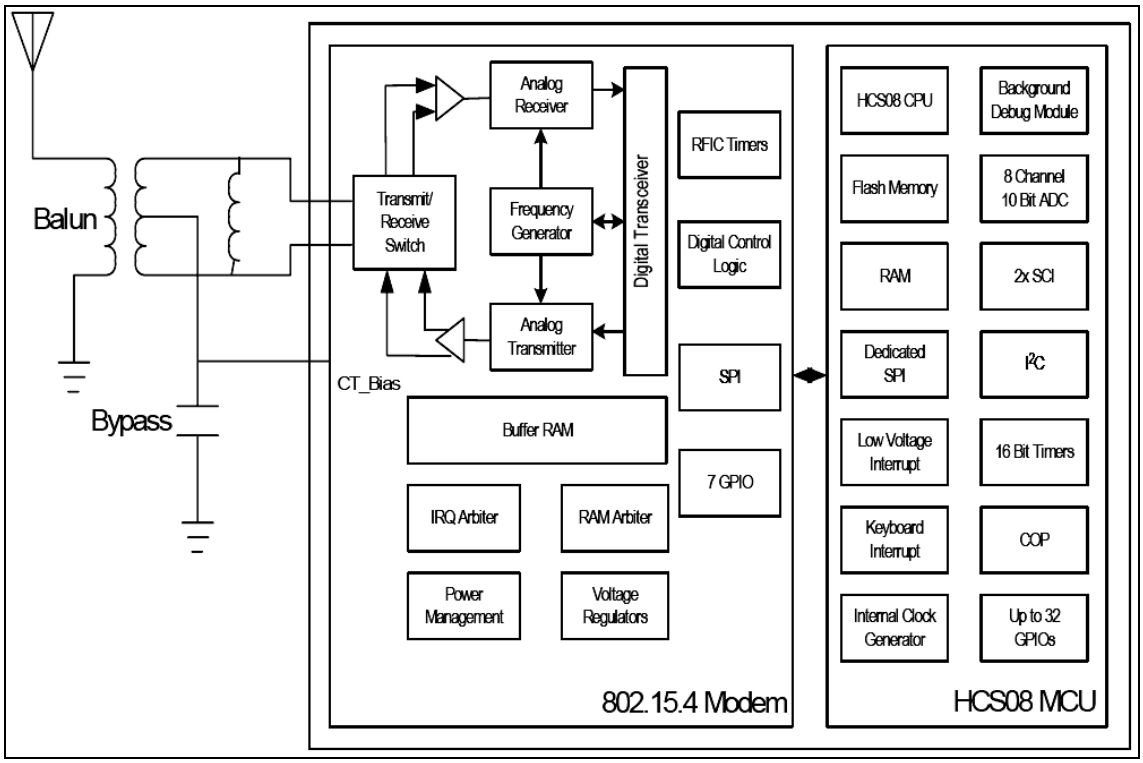


Diagrama de bloques de la plataforma MC13213 [33]

A la vista de esta figura, se pueden enumerar pues las características más representativas de la MCU contenida en la plataforma MC13213:

- Microcontrolador alimentado mediante voltajes de 2 a 3.4 voltios, que dispone de una CPU HCS08 de bajo consumo a 40 MHz.
- 60KB de memoria Flash (permite su borrado y regrabado en tiempo real mediante un único valor de alimentación) con capacidades de protección y securización de bloques. Permite hasta 100.000 ciclos de borrado y reprogramación en condiciones típicas. El módulo software NVM (explicado en el capítulo 4) permitirá emplear esta memoria Flash adicionalmente para el almacenamiento de datos de contexto no volátiles.
- 4KB de memoria RAM securizable.
- Múltiples modos de bajo consumo (3 modos de hibernación y uno de espera).
- Interfaz SPI conectado internamente al módem 802.15.4.
- Diversos temporizadores hardware de 16 bits altamente configurables.

- Puerto con capacidad de configurar interrupciones KBI (interrupciones de teclado).
- Conversor analógico-digital (ADC) de 8 canales configurables de 8 a 10 bits.
- 2 Interfaces para comunicaciones serie independientes. Permitirán configurar al menos una UART para la comunicación RS232 con el exterior.
- Múltiples fuentes de reloj posibles: generado a partir del reloj interno (243kHz), generado a partir de un cristal externo u oscilador, generado a partir del reloj del módem (muy preciso), mediante el reloj de arranque (~8MHz), etc.
- Interfaz I^2C a 100kbps de velocidad. Permitirá comunicar la plataforma MC13213 con la memoria auxiliar externa EEPROM integrada en el ND07.
- Potente interfaz hardware para la programación y depuración de código sobre el chip. Implementado mediante el módulo DBM (*Background Debug Module*).
- Mecanismos hardware para la protección del sistema: interrupción programable para la detección de caídas de la alimentación (*Low Voltage Interrupt*, LVI), temporizador *watchdog* (COP), detectores de comandos ilegales (que permitirán resetear el dispositivo vía software), etc.
- Hasta 32 líneas de entada/salida de propósito general configurables.
- Circuito de reset hardware.

De forma análoga, se pueden resumir las principales características del módem radio:

- Transceptor 100% compatible con el estándar 802.15.4 capaz de emitir datos a modulados mediante O-QPSK, una velocidad de 250kbps, sobre 16 canales de 5 MHz en la banda de 2,4GHz, y empleando técnicas de espectro ensanchado.
- Potencia de salida nominal de -1 a 0dBm, programable de -27dBm a 3dBm.
- Sensibilidad menor de -92dBm con un 1% de PER (ante paquetes de 20 bytes).
- Conmutador integrado para las líneas de transmisión y recepción.
- Posibilidad de configurar diferentes líneas para la transmisión y para la recepción de paquetes (funcionamiento dual). Posibilidad de acoplarle a esas líneas etapas de amplificadores de potencia y amplificadores de bajo ruido.
- Soporta tres modos de bajo consumo para el ahorro de baterías.
- Salida de reloj de frecuencia configurable (que habitualmente se empleará como fuente externa para la generación del reloj de referencia de la MCU).
- Modos configurables de emisión: chorro de bits (*stream*) y paquetes (*packet*).
- 7 líneas de entrada/salida de propósito general adicionales.

Puesto que los módulos anteriormente enumerados son demasiado numerosos como para que todos sean explicados en profundidad en esta memoria, se dedicarán los primeros apartados del capítulo 4 para la descripción detallada de sólo aquellos que estén directamente relacionados con el desarrollo del bootloader y del cargador RS232.

A partir de la descripción general de los módulos integrados en la plataforma MC13213, así como las características del ND07, parece más que evidente que este dispositivo será idóneo para el desarrollo y depuración de las aplicaciones de este proyecto, ya que:

- Dispone de una memoria auxiliar EEPROM, con capacidad suficiente como para almacenar imágenes *firmware* (ver apartado 4.1.3), que se comunicará con el microcontrolador mediante un bus I^2C (módulo que integra la plataforma MC13213).
- Dispone de suficiente memoria RAM y ROM como para que sea instalable en la mayoría de las aplicaciones ZigBee generables a parir de las librerías de la *BeeStack 1.0.5* (ver subapartados 3.6 y 3.7).
- La memoria no volátil del dispositivo permite su borrado y sobrescritura en tiempo de ejecución y las librerías de la *BeeStack 1.0.5* suministran un API con ese fin.
- El MC13213 soporta un mecanismo para la detección de comandos inválidos que resetea el dispositivo (otorgando la capacidad de reinicializar el ND07 mediante comandos software).
- Las librerías suministradas para 802.15.4 y ZigBee otorgan un considerable acceso a las capas inferiores del protocolo y a la secuencia de arranque del dispositivo (aunque no suministran las fuentes de código de las librerías) facilitando pues el desarrollo de una aplicación de bajo nivel altamente dependiente del hardware como es el bootloader.

Adicionalmente, es importante destacar que en el momento en que se comenzó el desarrollo de este proyecto, *Freescale* suministraba abundante documentación para todos los componentes hardware y software de esta plataforma, además de un servicio técnico vía web.

3.4.3. Configuración del Dispositivo (Funcionamiento y Desarrollo)

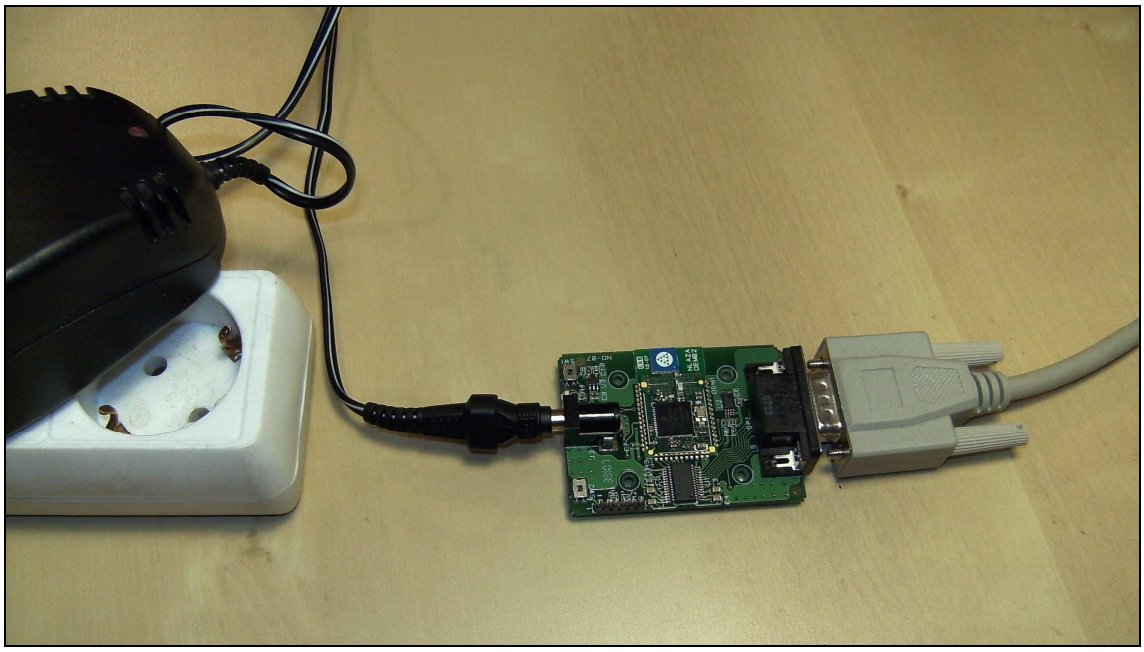
Una vez seleccionado el módulo ND07 como dispositivo ZigBee sobre el que se desarrollarán las aplicaciones del bootloader y del cargador RS232 (aunque todavía se justificará aún más esta decisión en los siguientes subapartados), será necesario estudiar las diferentes configuraciones de los dispositivos implicados que se observarán a lo largo del proyecto.

Así pues, cabe distinguir principalmente dos escenarios: funcionamiento normal del dispositivo (con las aplicaciones instaladas en el *firmware*) y funcionamiento de test (mientras dure el proceso de desarrollo y depuración de las aplicaciones en memoria).

El primer escenario presenta la configuración más sencilla: el ND07 se alimentará a través de la red eléctrica mediante un transformador a 4,5 voltios y se conectará su salida RS232 al puerto serie de un ordenador mediante el cable adecuado.

En este caso, la alimentación se suministrará al dispositivo a través de su conector específico accesible desde el exterior de su caja (*jack* “NEB 21 R” con pin de 1.95mm). Esta entrada estará protegida y soportará variaciones moderadas del valor de la alimentación.

La imagen siguiente muestra la configuración de los dispositivos implicados:



Escenario 1: Funcionamiento normal del ND07 (sin caja)

Este escenario será habitual durante las últimas fases del proceso de desarrollo y en instalaciones de campo reales.

Ocasionalmente se incluirá en el sistema un *sniffer* para la captura de tramas radio que permita comprobar el correcto funcionamiento del protocolo ZigBee programado en el ND07.

El segundo escenario será el que se observará más habitualmente a lo largo del proceso de desarrollo y requerirá de la inclusión de varios dispositivos adicionales que permitan la correcta alimentación, programación y depurado del ND07.

En primer lugar será necesario adquirir al menos un *USB MultiLink* (suministrado dentro de los kits de desarrollo de *Freescale*) e instalar sus *drivers* en el ordenador que se empleará para desarrollar las aplicaciones.

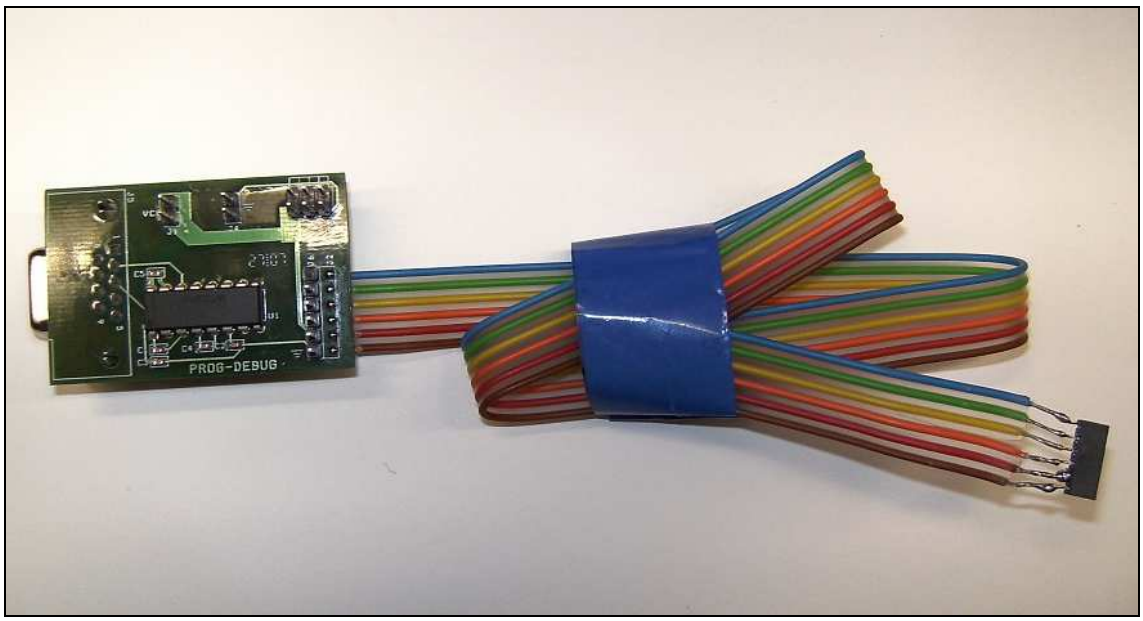
El *USB MultiLink* es un dispositivo que permitirá la programación y depuración de aplicaciones programadas para las placas de desarrollo de la plataforma MC13213 de *Freescale* a través de su módulo DBM.

USB MultiLink

Este dispositivo se combinará habitualmente con el depurador (*debugger*) integrado dentro del IDE *CodeWarrior for Microcontrollers* a para disponer de un entorno gráfico, potente y flexible, para la búsqueda y eliminación de errores.

Sin embargo, el *MultiLink* dispone de un conector que no es directamente compatible con el ND07 y que requiere pues de un dispositivo intermedio.

Este dispositivo, llamado programador/depurador *NLaza-Freescale*, requerirá a su vez de una alimentación externa que se logrará acoplando una fuente de alimentación regulada a 2,7 voltios.



Programador/Depurador *NLaza-Freescale*

De este modo, siempre que se desee programar o depurar aplicaciones del ND07, será necesario conectar el programador/depurador *NLaza-Freescale* a los pines de la placa (PCB),

accesibles sólo mediante la apertura de la caja del dispositivo. El programador/depurador se conectara a su vez a un *USB MultiLink* y se alimentará mediante a 2,7 voltios constantes.

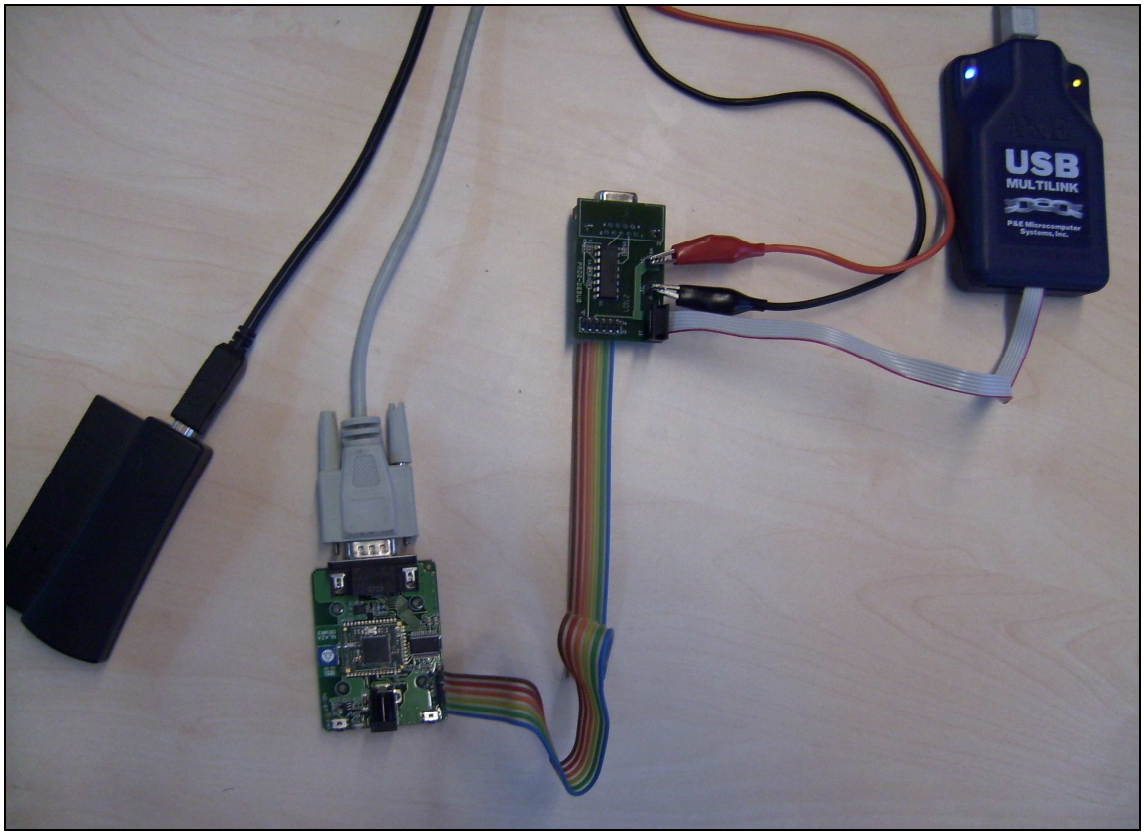
Desafortunadamente, con el objetivo de minimizar el número de componentes del ND07 (y por tanto el precio total del dispositivo), la placa no dispone de ningún tipo de circuito de protección desde los pines ni de un mecanismo de regulación de tensión adecuado.

Así pues, es imprescindible respetar la polaridad de los pines del programador/depurador y mantener la alimentación externa en torno a los 2,7 voltios (aproximadamente) o de lo contrario es posible dañar el dispositivo de forma permanente.

De forma análoga al primer escenario, para el correcto funcionamiento de la aplicación del cargador RS232, será necesario conectar la salida DE-9 hembra del ND07 al puerto serie de un ordenador.

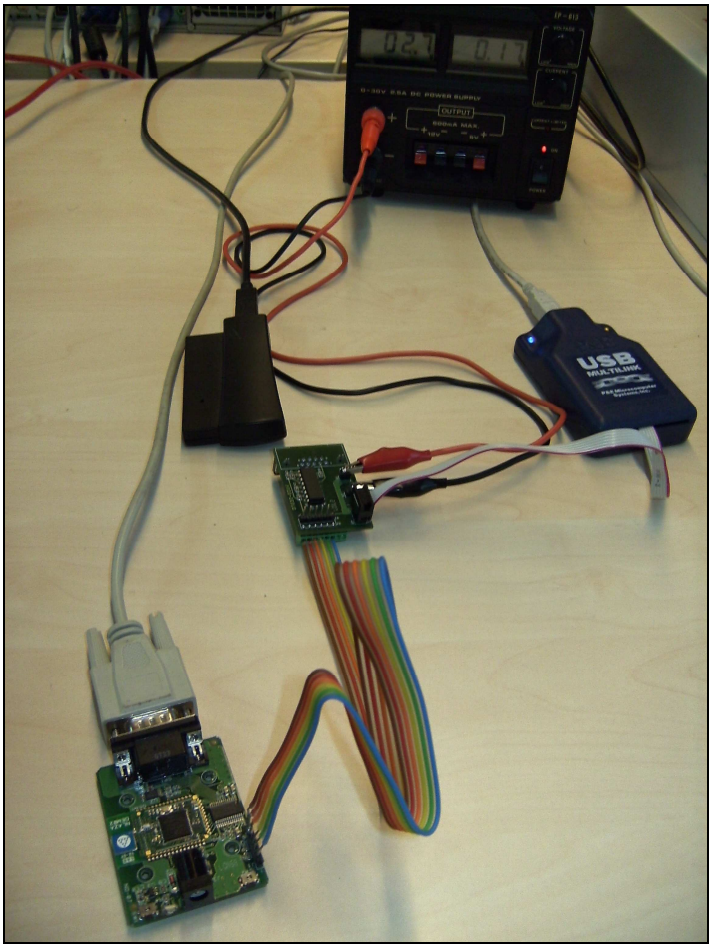
Una vez este escenario esté preparado, el *CodeWarrior* permitirá programar y depurar aplicaciones sobre el ND07 de forma potente y versátil.

Las siguientes imágenes muestran la configuración y conexión de los dispositivos habitualmente implicados en el segundo escenario:



Escenario 2: Desarrollo y depuración (serie y radio)

De igual forma que en el primer escenario, ocasionalmente se incluirá un *sniffer* radio para capturar las tramas emitidas por el dispositivo y así evaluar su correcto funcionamiento.



Escenario 2: Desarrollo y depuración (serie y radio)

Finalmente, es necesario hacer hincapié en que ambos escenarios son incompatibles ya que alimentar los dispositivos implicados simultáneamente mediante dos mecanismos diferentes (red a 220 voltios y fuente de alimentación continua a 2,7 voltios) puede dañar los equipos de forma permanente.

3.5. Entorno de Desarrollo: CodeWarrior for MicroControllers v5.1

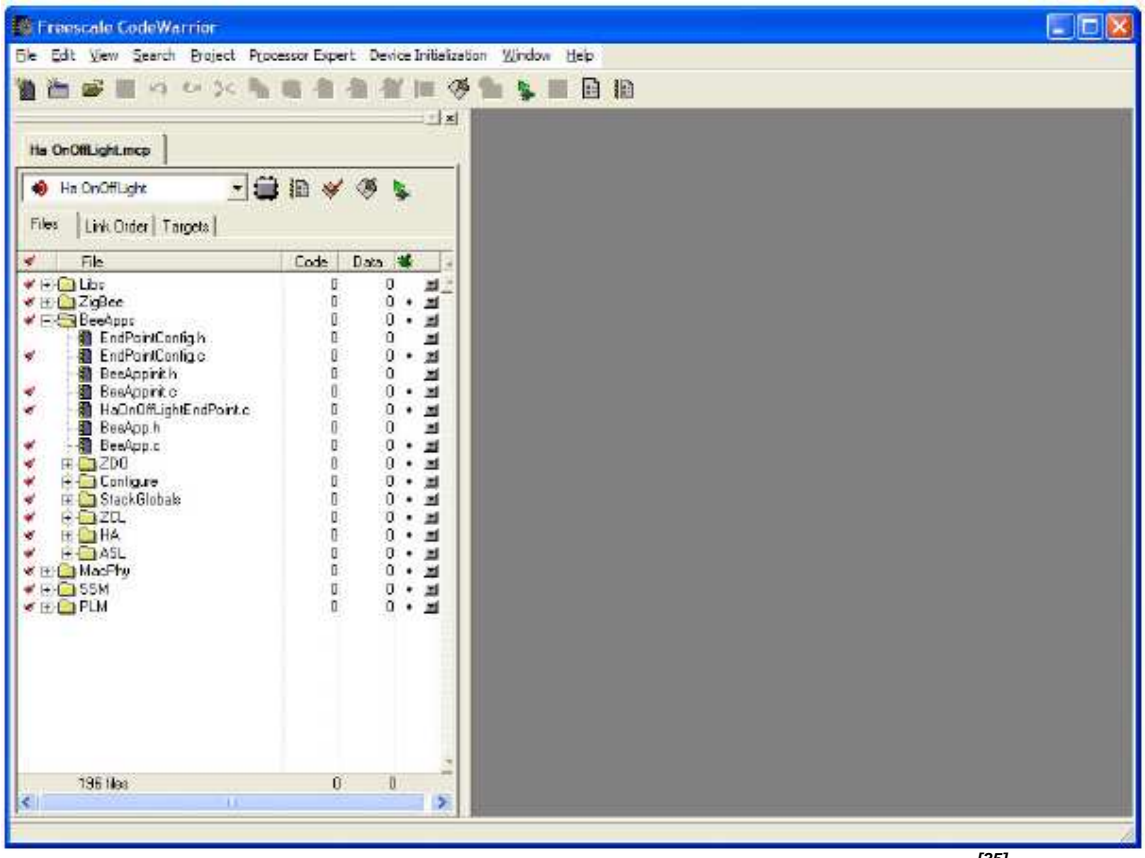
El *CodeWarrior* es el entorno de desarrollo integrado (IDE) suministrado por *Freescale* para la programación de aplicaciones destinadas a funcionar sobre los microcontroladores de la familia HC(S)08 [34].

La plataforma MC13213 integrada en el ND07 pertenece a dicha familia, de modo que será el *CodeWarrior* el IDE elegido para desarrollar los *firmwares* de este proyecto.

Como todo entorno de desarrollo que se precie de serlo, el *CodeWarrior for MicroControllers* incluye un conjunto de herramientas imprescindibles para trabajar de forma eficiente:

- Un compilador configurable para microcontroladores de la familia HC(S)08.
- Un ensamblador (*assembler*) y un enlazador (*linker*) considerablemente configurables
- Un depurador (*debugger*) capaz de programar la memoria flash de dispositivos y controlar la ejecución de sus aplicaciones con el fin de buscar y corregir errores. Para su correcto funcionamiento es necesario instalar previamente un periférico programador/depurador llamado *USB MultiLink*.
- Un entorno de programación amigable con potentes opciones de búsqueda de funciones, adecuada navegación entre ficheros del proyecto y con coloreado de sintaxis para código en C (*syntax highlighting*).

La siguiente captura de pantalla muestra el entorno de desarrollo tal como se presenta a un programador de aplicaciones destinadas a la plataforma MC13213:



Captura de pantalla del IDE *CodeWarrior for Microcontrollers v5.1.* [35]

El CodeWarrior dispone además de la capacidad de importar proyectos generados con la herramienta *BeeKit* (ver apartado siguiente), de forma que es posible comenzar a programar *firmware* de dispositivos a partir de aplicaciones de ejemplo previamente configuradas en dicha

herramienta. Esto acelerará drásticamente el proceso de desarrollo y reducirá considerablemente el *time-to-market* del producto.

Esta propiedad del CodeWarrior se empleará también en este proyecto ya que la aplicación del Cargador RS232 (ver capítulo 4.2) se programará modificando una aplicación de ejemplo de interfaz combinado (*HA Combined Interface*) generada con el *BeeKit*.

A lo largo del desarrollo de este proyecto, el IDE del *CodeWarrior* también ha ido evolucionando, partiendo desde la versión 3.1 (plagada de errores) hasta la versión 6.2, la última publicada.

Sin embargo, las versiones posteriores a la versión 5.1, otorgaban soporte para muchos nuevos chips y las licencias adquiridas no eran compatibles. De este modo, la mayor parte del tiempo se ha trabajado con la versión 5.1 que, aún presentando algunas limitaciones y errores, es un IDE bastante potente y robusto.

Es necesario remarcar que el CodeWarrior es un IDE considerablemente flexible y robusto, mejor que otros entornos evaluados de la competencia.
Por ejemplo, el IDE suministrado por *Ember* para desarrollar aplicaciones para su chip EM250 (que sería el equivalente aproximado al MC13213 en cuanto a prestaciones) es con diferencia mucho menos amigable para el programador.

Este entorno de *Ember*, llamado *Xide*, carece de las funcionalidades básicas necesarias para la navegación sencilla entre funciones y ficheros y además su compilador es menos de la mitad de eficiente que el incluido en el *CodeWarrior* (de modo que un *firmware* requerirá del doble de memoria flash para ser almacenado en el caso de *Ember* que en el caso de *Freescale*).

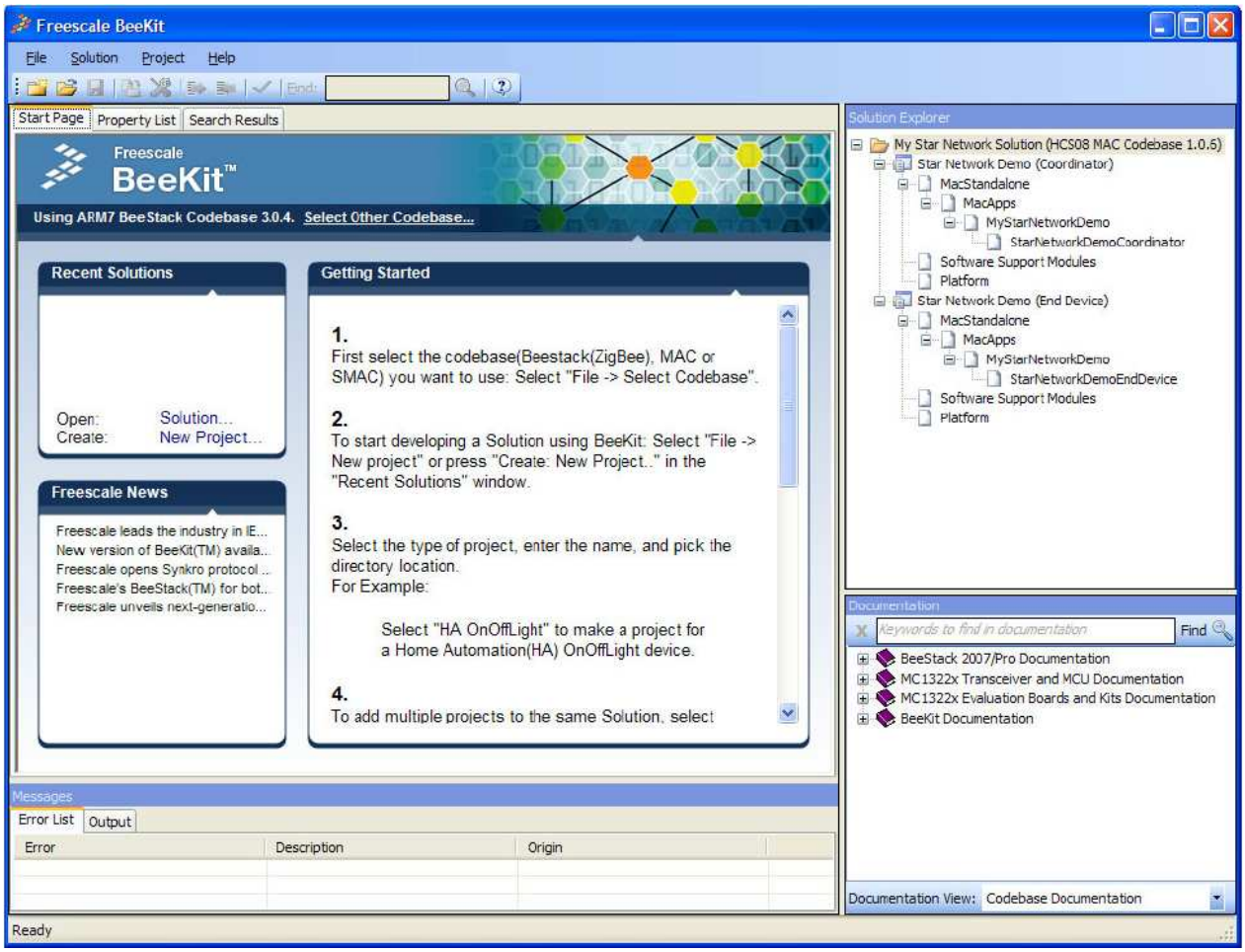
Así pues, se concluye que la herramienta *CodeWarrior for MicroControllers v5.1* cumplirá todos los requisitos de robustez, potencia y flexibilidad necesarios para desarrollar adecuadamente las aplicaciones objetivo de este proyecto.

3.6. Generación de Aplicaciones ZigBee de Ejemplo: Beekit

BeeKit es una herramienta que provee de un interfaz gráfico de usuario (GUI) mediante el cual un usuario puede crear, modificar, guardar y actualizar soluciones inalámbricas basadas en las pilas de protocolos suministradas por *Freescale*.

Una de sus características más atractivas es que incluye un asistente que permite al usuario configurar rápidamente los parámetros de una aplicación antes de generar el proyecto,

reduciendo considerablemente la necesidad de navegar entre ficheros individuales configurándolos manualmente.



Captura de pantalla de la ventana principal del *BeeKit Wireless Connectivity Toolkit* [36]

Mediante el amplio repositorio de librerías de comunicaciones inalámbricas, plantillas y aplicaciones de ejemplo que el *Beekit* incluye, el usuario podrá generar los ficheros adecuados para su entorno de trabajo que serán después importados desde el IDE del *CodeWarrior* para continuar su desarrollo y depurado.

Los proyectos de ejemplo generados con esta herramienta suelen ocupar entre 45KB y 55KB de memoria Flash y unos 3,5KB de memoria RAM, dependiendo de la configuración elegida.

Esto significa que el desarrollador dispondrá habitualmente de menos de 1KB de memoria RAM y de 5KB a 15KB de memoria Flash para programar sus modificaciones de la aplicación principal (incluyendo el bootloader, para el caso de este proyecto).

BeeKit es una aplicación fácilmente escalable diseñada para dar soporte a nuevas librerías y funcionalidades según se van publicando.

Aunque la descarga de esta aplicación incluye una licencia para el uso ilimitado de las librerías de *SMAC* y 802.15.4, la licencia de la pila *BeeStack* (para ZigBee, ver apartado siguiente) caducará a los 90 días y será necesario adquirirla si se desea seguir utilizando la herramienta.

No obstante, puesto que *BeeKit* es una herramienta ideada para acelerar el desarrollo continuo de nuevas aplicaciones, no debería ser necesario su uso una vez se haya configurado y generado la aplicación "plantilla" sobre la que se desarrollará el bootloader y el cargador RS232. Esta "plantilla" será, a efectos de este proyecto, un ejemplo de un coordinador ZigBee programado sobre una aplicación de interfaz combinado, del perfil de automatización del hogar (*HA Combined Interface*).

Desafortunadamente, a lo largo del proceso de desarrollo del bootloader, se han sucedido actualizaciones críticas de la librería *BeeStack* que sólo eran accesibles mediante la descarga e instalación completa del *BeeKit*, de forma que si se deseaba corregir los errores e incluir las posibles mejoras era de nuevo imprescindible adquirir una licencia.

De este modo, a lo largo del desarrollo de este proyecto ha sido necesario instalar las sucesivas versiones del *BeeKit* según se iban publicando para así evaluar las nuevas librerías de 802.15.4 y ZigBee que incluía. Esto ha representado por lo menos una actualización cada dos meses desde principios del 2007 (versión 1.0.0) hasta principios del 2008, cuando *Freescale* dejó de publicar revisiones de la pila de ZigBee 2006 (ver apartado 4.1.6).

La última revisión publicada a día de hoy es la versión 1.9.10 (septiembre 2009), e incluye soporte para las librerías de los protocolos SMAC, ZigBee, ZigBee PRO, RF4CE y *SynkroRF*, además de para los perfiles *Home Automation* y *Smart Energy*.

3.7. Librería ZigBee 2006 de *Freescale*: *BeeStack* 1.0.5.

3.7.1. Introducción

Tras numerosos retrasos en su publicación, la especificación pública ZigBee 2006 finalmente vio la luz en diciembre de ese mismo año. Desafortunadamente, aún fue necesario esperar varios meses para disponer de las primeras librerías ZigBee 2006 compatibles con el chip MC13213 de *Freescale*: la *BeeStack* 1.0.0.

Sin embargo, las librerías 802.15.4 para dicha plataforma llevaban ya más de un año en el mercado y sirvieron para ir familiarizándose con las capas que lo conforman (es necesario recordar que ZigBee 2006 se basa en el nivel físico y de acceso al medio del estándar IEEE 802.15.4-2003) y con el entorno de desarrollo común *CodeWarrior for Microcontrollers*.

A principios del año 2007 se publicaron pues la primera versión de las librerías y, sólo un par de meses más tarde, su primera revisión, la *BeeStack 1.0.1.*

Fue poco después cuando *Freescale* decidió incluir las actualizaciones de la librería ZigBee 2006 dentro de una nueva herramienta diseñada para generar aplicaciones de ejemplo de automatización del hogar mediante ZigBee 2006 y su protocolo propietario, SMAC. Dicha herramienta recibiría el nombre de *BeeKit* (ver apartado anterior) y requeriría de una licencia adicional para poder acceder a las actualizaciones.

De este modo, a partir de mediados del año 2007, *Freescale* publicaría una revisión de las librerías 802.14.5 y ZigBee cada 4 meses aproximadamente para las que era necesario descargarse e instalar una nueva versión del *BeeKit*, generar un proyecto de una aplicación de ejemplo similar a la que se estaba empleando, portar sus librerías y comprobar que el API no había sufrido cambios significativos (desafortunadamente, esto no siempre sucedería).

Este último paso requeriría del empleo de aplicaciones de comparación de ficheros como el *WinMerge* [20] y de una considerable comprensión del funcionamiento de todo el API de la *BeeStack* para poder aplicar correctamente las actualizaciones del código.

De este modo, la *BeeStack* sufriría modificaciones de mayor o menor importancia hasta poco antes del verano del año 2008, cuando se averiguaría a través de conversaciones con el servicio técnico de *Freescale*, que la compañía abandonaba todo soporte a las librerías ZigBee 2006 para destinar sus esfuerzos al desarrollo de las librerías de ZigBee PRO (*BeeStack 2.0.0* en adelante).

De este modo, la *BeeStack 1.0.5* (última revisión obtenida) será la versión más estable publicada de las librerías ZigBee 2006 para el chip MC13213 y, por tanto, la base sobre la que se fundamentará este proyecto.

En el siguiente subapartado se describirá pues la arquitectura de capas que conforman la *BeeStack 1.0.5* y el API suministrado para la programación del bootloader y el cargador RS232, objetivos finales de este proyecto.

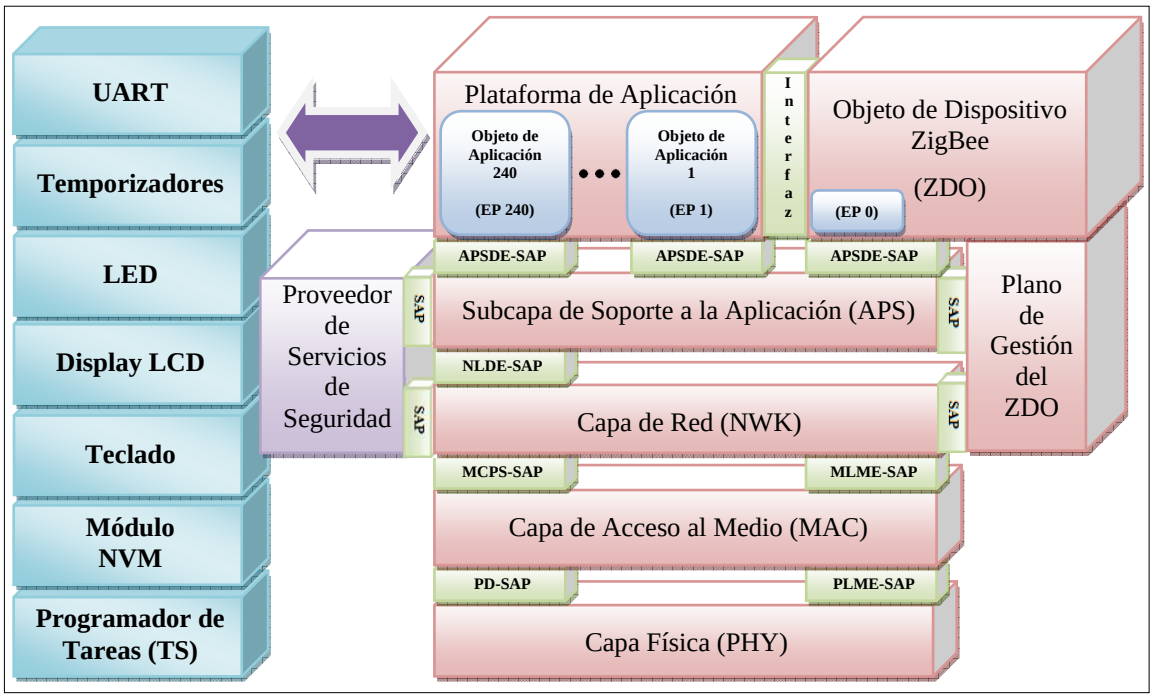
En último lugar se incluirá una descripción de los errores encontrados en las librerías a lo largo del desarrollo del proyecto y los problemas que éstos acarrearán.

3.7.2. Descripción de Capas y API

Según *Freescale*, el término *BeeStack* se empleará para describir todo el software que se programará en los dispositivos ZigBee destino a excepción del código específico de la aplicación.

La pila *BeeStack* está formada principalmente por componentes de red, destinados a establecer la conexión y la comunicación entre dispositivos ZigBee, y por componentes específicos de la plataforma, que proveerán a la aplicación de un interfaz de operación y acceso al hardware del equipo.

La siguiente figura muestra los módulos que conforman la pila [35]:



Componentes de la librería ZigBee *BeeStack 1.0.5.*

El funcionamiento ordenado de todos los módulos que componen la *BeeStack* se basa principalmente en la gestión que realiza un programador de tareas (*Task Scheduler*, TS) que se suministra en código fuente para posibilitar su edición por el desarrollador.

De este modo, la librería cuenta con que el programador de tareas se encargue de repartir el tiempo de ejecución de forma equitativa (sin dejar ningún módulo en estado de inanición) pero dando mayor prioridad a las capas y componentes más críticos para el correcto mantenimiento del dispositivo dentro de la red (por lo general los niveles más cercanos a la capa física suelen ser los más prioritarios).

De forma muy parecida pues a la que se definen las diferentes capas funcionales en la especificación ZigBee 2006, se pueden distinguir las siguientes tareas dentro de la librería *BeeStack 1.0.5*:

La tarea encargada de la capa de red, responsable del encaminamiento de paquetes, del descubrimiento de rutas, de la transmisión de tramas *unicast* o *broadcast*, del envío *unicast* o *broadcast* y del descarte de los datos cuyo destino no sea el propio dispositivo o la red a que pertenece.

La tarea encargada de la subcapa de soporte a la aplicación (APS), responsable de la entrega y recepción de datos de aplicación, del establecimiento de *bindings* entre distintos *endpoints* y del asentimiento de paquetes extremo a extremo. Adicionalmente se ocupa de realizar todo el proceso de autenticación en redes securizadas, incluyendo la gestión del centro de confianza en los nodos que sean coordinadores de red ZigBee.

La tarea encargada de la plataforma de aplicación (AF), responsable de la entrega de indicaciones de datos (*data indications*) y confirmaciones (*data confirms*) a los distintos *endpoints* de las aplicaciones del nodo, según se vayan recibiendo.

La tarea encargada del objeto de dispositivo ZigBee (ZDO), encargada de mantener y gestionar el estado del dispositivo en la red, así como de los procedimientos de asociación o abandono ordenado de ésta.

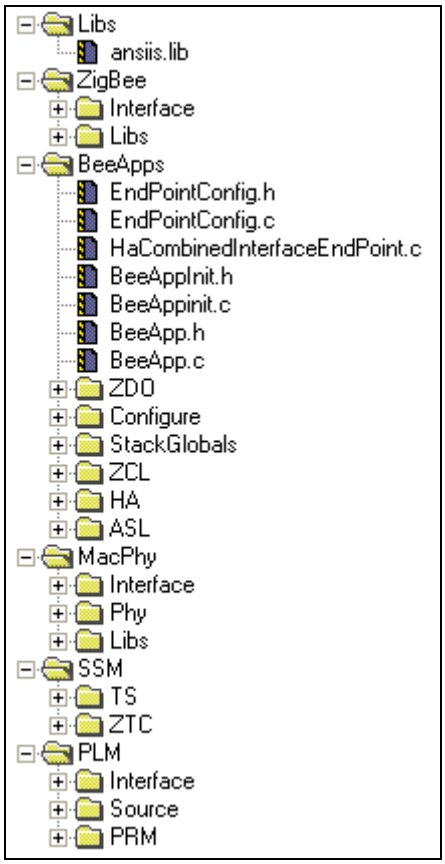
Por otro lado, además de las tareas asociadas a las capas de la especificación ZigBee, la librería incluye por defecto otras tareas fundamentales para el correcto funcionamiento del protocolo y de los módulos integrados en el chip MC13213:

La tarea del perfil del dispositivo ZigBee (ZDP), encargada de tratar las peticiones y respuestas para una serie de comandos comunes a todos los dispositivos ZigBee y destinados a gestionar los diversos nodos dentro de la red (por ejemplo consultas de direcciones, servicios disponibles, *endpoints* activos, etc.).

Las tareas relacionadas con la gestión de los componentes de la plataforma (PLM), encargadas de interactuar con los elementos hardware como pulsadores (mediante *polling* o interrupciones KBI), LEDs (mediante encendidos, apagados, parpadeos, etc.), *displays* numéricos o gráficos (gestión de LCDs) o temporizadores (TMR). Todos los componentes de la PLM pueden ser modificados para cumplir las especificaciones impuestas por el desarrollador.

Así pues, el árbol de directorios y ficheros que compone un proyecto basado en las librerías de la BeeStack 1.0.5 también puede describirse de forma sencilla atendiendo a una clasificación funcional por capas y módulos.

Para realizar dicha descripción nos basaremos en el árbol de directorios generado por el *BeeKit* al exportar un proyecto de ejemplo de una aplicación de Interfaz Combinado del perfil de automatización del hogar (*HA Combined Interface*).
Esta aplicación de ejemplo será la base desde la que se partirá para programar el bootloader y el cargador RS232 de este proyecto. La figura siguiente muestra pues el árbol de directorios de una aplicación ZigBee 2006 habitual, basada en la librería *BeeStack 1.0.5*:



Directorios y ficheros de la aplicación "*HA Combined Interface*" (*BeeStack 1.0.5*)

Como puede observarse a primera vista, un proyecto ZigBee 2006 puede dividirse en los siguientes bloques funcionales: *Libs*, *ZigBee*, *BeeApps*, *MacPhy*, *SSM* y *PLM*.

El directorio *Libs* contiene únicamente una librería necesaria para el funcionamiento de cualquier aplicación programada en C para este chip.

El directorio *ZigBee*, como su nombre indica, contiene la librería ZigBee 2006 específica del tipo de dispositivo elegido (dispositivo final, router o coordinador) y los interfaces (API) entre la

capa de aplicación y el resto de capas de la arquitectura ZigBee (red, soporte de aplicación, ZDO, etc.).

El directorio *BeeApps* contendrá pues todos los ficheros relacionados con la aplicación principal que se ejecutará en el dispositivo. De este modo incluirá todo el código relacionado con los *endpoints*, el ZDO, la *ZigBee Cluster Library* (ZCL, necesaria para los perfiles), con el tipo de dispositivo elegido dentro del perfil de automatización del hogar (*HA*) y con la aplicación ZigBee principal en general.
El directorio *MacPhy* contendrá las librerías y el API correspondientes a las capas de acceso al medio (MAC) y física (PHY) basadas en 802.15.4 que serán necesarias para el correcto funcionamiento del protocolo.

El directorio *SSM* contendrá el programador de tareas (*TS*) y el interfaz necesario para poner en marcha una herramienta especial de *Freescale* empleada para depurar el funcionamiento de las comunicaciones entre las diferentes capas de la *BeeStack*: la *ZigBee Test Client* (*ZTC*).

Finalmente, el directorio *PLM* contendrá todos aquellos ficheros que estén relacionados con la gestión de un módulo del chip, un componente hardware, o un periférico del dispositivo. De este modo, este directorio incluirá el API para: la *UART*, los temporizadores software, el módulo *NVM* (para almacenamiento de datos no volátiles), el módulo de bajo consumo (*LPM*), los *LEDs*, los pulsadores (*Keyboard*), el *display* (*LCD*) así como todos los parámetros para su correcta inicialización y configuración.

Adicionalmente, este directorio incluirá todos los ficheros relacionados con el arranque del dispositivo, de modo que un estudio en profundidad de su código será necesario para la correcta inclusión del bootloader dentro de la secuencia de arranque del equipo.

Puesto que el API para establecer, asociarse y mantener una red ZigBee está debidamente incluido en la *BeeStack*, la función principal de un desarrollador de aplicaciones ZigBee habitualmente consistirá en modificar tan sólo algunos ficheros del directorio *BeeApps* y del directorio *PLM* y probablemente incluir algunos nuevos. Rara será la ocasión en que un programador de aplicaciones necesite modificar ficheros incluidos dentro de los directorios *Libs*, *ZigBee*, *MacPhy* o *SSM*.

Desafortunadamente, aunque el cargador RS232 realmente no requerirá trabajar con directorios adicionales, el bootloader es lo suficientemente “intrusivo” como para que sin requerir de la modificación de ficheros alojados en directorios diferentes al *PLM* o al *BeeApps*, sí que requerirá de un conocimiento profundo acerca del funcionamiento de todos sus módulos para evitar cualquier tipo de colisión (ver capítulo 4 para las conclusiones sobre cada módulo).

Es necesario recordar que el bootloader no se comportará nunca como un programa de la capa de aplicación ZigBee, sino como un componente del propio “sistema operativo” que se ejecutará antes incluso que el programador de tareas.

El API de comunicaciones ZigBee incluido en la *BeeStack 1.0.5* es muy extenso y se sale del ámbito de esta memoria. No obstante será necesario disponer al menos de unas nociones acerca de su funcionamiento para la programación del cargador RS232, para lo cual se recomienda la lectura de las referencias **[35]**, **[37]** y **[38]** de la bibliografía.

3.7.3. *Bugs* Conocidos y Reconocidos por *Freescale*

Desde la primera librería de ZigBee 2006 que publicó *Freescale* para el chip MC13213 a principios del año 2007 hasta que dejó de darle soporte a mediados del 2008, se han ido detectando y subsanando un considerable número de errores críticos de la pila que hacían que ésta no cumpliese la especificación ZigBee (paradójicamente todas sus versiones han obtenido la certificación ZigBee 2006).

A lo largo de este proyecto se detectarán casi una decena de errores de las librerías y de las aplicaciones de ejemplo suministradas por *Freescale*. De este modo el escenario de aplicación del bootloader habrá de considerarse como un entorno en constante supervisión y perfeccionamiento, con la inestabilidad ocasional que eso en ocasiones representa.

Será necesario pues familiarizarse con el servicio técnico de *Freescale,* con el que habrá que ponerse en contacto en numerosas ocasiones para identificar, reportar y obtener soluciones de contingencia para los errores detectados.

Aunque muchos de estos errores eran de extrema gravedad para el correcto funcionamiento del protocolo ZigBee y su compatibilidad con otros dispositivos que cumpliesen el estándar, sólo se ha descubierto uno de ellos que afectase directamente al bootloader.

Este error grave consistirá en que el envío de tramas radio con asentimiento (ACK) desde la aplicación provocará que un *flag* interno de la pila se quede activado indefinidamente.

Este flag (denominado *mNvCriticalSectionFlag* dentro de las librerías) está diseñado para bloquear la actualización periódica automática de los datos no volátiles almacenados en el módulo NVM (de estar habilitado dicho componente) mientras tenga lugar cualquier otra operación prioritaria que consuma muchos recursos. El componente NVM se explicará más adelante en el apartado 4.1.2.2.5.

Esto se debe a que la operación de actualización de los datos almacenados en la NVM es una operación lenta y crítica, que no debe interrumpirse en la medida de lo posible.

Al provocarse este error, y el flag nunca desactivarse, las variables almacenadas en la NVM nunca se actualizarán y el dispositivo será incapaz de mantener eficientemente su contexto dentro de la red (direcciones de hijos, asociaciones, etc.).

De este modo el almacenamiento del *flag* de actualización del bootloader, que idealmente se habría localizado dentro de las páginas reservadas para la NVM (cosa más que lógica), tendrá que portarse a la EEPROM. Esto representará una considerable pérdida de eficiencia ya que será imprescindible la inicialización del API de comunicación I^2C durante el arranque del dispositivo para consultar el *flag* de regrabación en EEPROM, cuando de haber estado en la NVM su lectura habría resultado inmediata.

Este problema (que a fin de cuentas tiene una solución sencilla), fue considerablemente difícil de detectar por la aleatoriedad de su aparición y representó una considerable cantidad de tiempo perdido durante el proceso de depurado del bootloader aplicado a situaciones reales (con dispositivos ZigBee funcionando en red). La ineficiencia y lentitud del servicio técnico de *Freescale* (ver apartado 4.1.5) también influyeron negativamente en la detección del error.

Aunque este error fue descubierto fruto del desarrollo de este proyecto y se notificó a *Freescale* a través de su servicio técnico, éste tardó muchos meses en reproducir el problema y reconocerlo públicamente. Poco después se anunciaba el cese de soporte a las librerías ZigBee 2006 por parte de *Freescale* para centrar sus esfuerzos en las nuevas librerías de ZigBee PRO. Probablemente sea ésta la razón por la que la última versión de las librerías ZigBee 2006 publicadas, la *BeeStack 1.0.5*, carezca del parche necesario.

Así pues, para las aplicaciones ZigBee 2006 de *Freescale*, la NVM será siempre un recurso inestable de dudosa fiabilidad. Aunque una vez migrado el *flag* de regrabación a la EEPROM este *bug* dejará de ser un problema a efectos del bootloader, sí que lo seguirá siendo para todas las aplicaciones ZigBee 2006 que transmitan paquetes radio con asentimientos, como es el caso del cargador RS232 de este proyecto.

De este modo, siempre que se diseñe una aplicación que emplee las librerías de la *BeeStack* 1.0.5 o inferiores, será necesario elegir entre la transmisión de asentimientos o el almacenamiento de datos no volátiles mediante el módulo NVM.

Suponiendo que las librerías ZigBee PRO incluyesen el parche buscado, sería necesario migrar las aplicaciones a nueva pila (*BeeStack 2.0.0* en adelante) la cual, desafortunadamente,

precisa de tanta memoria RAM y Flash que no suele haber suficiente disponible en el MC13213 (a menos que la aplicación ZigBee sea muy sencilla, incapaz de enrutar paquetes).

Estas nuevas librerías se han programado evidentemente pensando en otros chips más modernos de *Freescale* que disponen de mayor capacidad, de forma que los dispositivos que integren el chip MC13213 sólo podrán actuar como ZEDs (dispositivos finales, que requieren de mucha menos memoria) si desean funcionar dentro de una red ZigBee PRO 100% compatible sin *bugs*.

No obstante, la evaluación inicial de dichas librerías ZigBee PRO y la lectura de los foros de discusión de *Freescale* sugiere que su publicación ha sido demasiado precipitada (y por ello de baja calidad) debido principalmente a razones comerciales. Esta apreciación parece confirmarse cuando se observa que en menos de un año se han publicado demasiadas revisiones de la librería, migrando desde la *BeeStack 2.0.0* hasta la *BeeStack 3.0.5* (septiembre 2009).

De este modo se puede concluir que, debido a la naturaleza de las librerías ZigBee de *Freescale*, que proporcionan un API para un protocolo novedoso aún en constante evolución, el proceso de desarrollo de las aplicaciones requeridas para este proyecto implicarán no sólo la programación de código, sino también la frecuente detección, aislamiento, notificación y corrección de *bugs* dentro de las pilas suministradas.

3.8. Bus I²C (*Inter-Integrated Circuit*) y TWI (*Two Wire Interface*)

I²C es un bus de comunicaciones serie desarrollado por Philips en 1992 capaz de alcanzar velocidades hasta 3.4 Mbit/s (aunque su modo estándar, el más ampliamente extendido, proporciona una velocidad de 100Kbit/s). Es importante resaltar que esta velocidad de bit no será la velocidad de transferencia efectiva de datos, que puede llegar a ser menos de la mitad, ya que también se transmiten las cabeceras del protocolo, los bits de asentimiento, las direcciones de dispositivo y registro destino, etcétera.

Este bus está muy extendido en la industria, que lo emplea principalmente para comunicar microcontroladores con sus periféricos en sistemas integrados (*Embedded Systems*) o incluso para comunicar entre sí circuitos integrados que coexisten en la misma placa de silicio. Éste último caso será justo uno de los principales objetivos de este proyecto: comunicar mediante un bus I²C la plataforma MC13213 y la memoria EEPROM AT24C512 integradas ambas en la placa del dispositivo ZigBee NLaza ND07. Esta comunicación habrá de permitir la lectura de las imágenes de *firmware* almacenadas en la EEPROM con el fin de actualizar la aplicación que se encuentra en ejecución.

I²C es la base de otros protocolos como pueden ser:

- *ACCESS.bus*
- El interfaz *VESA Display Data Channel* (DDC)
- *System Management Bus* (SMBus)
- *Power Management Bus* (PMBus)
- *Intelligent Platform Management Bus* (IPMB)

Estos buses se basan en I²C pero definen algunos de sus parámetros y comandos de forma específica para el objetivo que quieren lograr. Es habitual que trabajen con rangos diferentes de voltaje y frecuencia e incluso empleen líneas adicionales de interrupción.

3.8.1. Diseño de Referencia

I²C emplea dos líneas para transmitir la información: una para los datos (línea **SDA**) y otra para la señal de reloj (línea **SCL**). Este bus emplea adicionalmente una tercera línea como referencia del sistema, que será tierra (línea GND). En casos como en el ND07, en que los componentes integrados que desean comunicarse mediante el bus I²C comparten la misma placa (y masa), esta tercera línea no será necesaria.

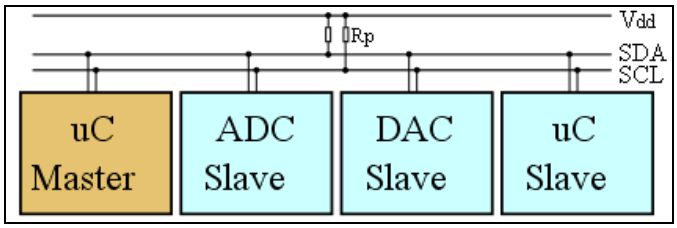
Las líneas SDA y SCL son salidas de drenador abierto, lo cual significa que necesitan una resistencia de polarización en el drenador para funcionar. Esto suele ser muy útil en situaciones

en que se pretende comunicar circuitos lógicos cuyos niveles altos son distintos. En el caso del ND07, su circuito integra una resistencia de pullup de 10KΩ para cada línea hasta la tensión regulada V_{cc}=2.5V.

El número máximo de dispositivos que pueden estar conectados simultáneamente al mismo bus está limitado por el espacio de direcciones y por la capacitancia total de la línea que no debe de superar los 400 pF.

En el caso del ND07 esto no supone ninguna limitación ya que sólo existen dos integrados conectados vía I^2C, el microcontrolador y la memoria EEPROM, y se encuentran separados tan sólo unos centímetros.

El siguiente esquema ilustra el esquema de un sistema interconectado mediante bus I^2C:



Ejemplo de conexionado de un bus I^2C con resistencias de *pull-up* [39]

Los dispositivos conectados al bus I^2C suelen funcionar como maestros o como esclavos, o incluso alternar dichos comportamientos (si el dispositivo posee esa funcionalidad) y para ello disponen de una dirección única que los identifica unívocamente (de 7 bits en las versiones habituales y hasta de 10 bits en las versiones más modernas del bus). Adicionalmente, I^2C es un bus multimaestro, lo que significa que permite la existencia de múltiples dispositivos conectados al mismo bus empleando el rol de maestro.

Un dispositivo, ya sea maestro o esclavo (o pueda funcionar con cualquiera de los dos roles), intercalará a lo largo de su vida útil dos modos posibles de funcionamiento: modo transmisión y modo recepción.

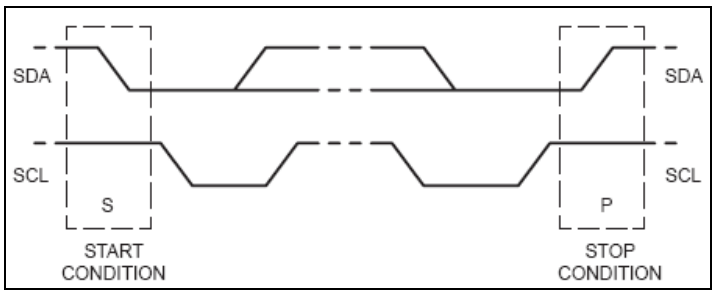
Cuando un dispositivo desea iniciar una comunicación, se convierte en maestro (en modo transmisión), genera la señal de reloj sobre la línea SCL y transmite los datos sobre la línea SDA. El receptor de esta transferencia será pues, un dispositivo esclavo funcionando en modo recepción.

Para iniciar una transmisión, un dispositivo maestro enviará por la línea SDA un patrón llamado "*start condition*" (bit **START**, explicado más adelante) seguido de los 7 bits de dirección del dispositivo (en casos de direcciones de sólo 7 bits, como en el ND07) con el que se quiere comunicar y luego un bit adicional que indicará su propósito de escribir (a 0) o leer (a 1) del esclavo.

Si existe un dispositivo con esa dirección, éste responderá con un bit **ACK** (un bit a 0) en el siguiente nivel alto de reloj (línea SCL). Esto provocará que el dispositivo maestro continúe en modo transmisión (si ha solicitado una escritura sobre el dispositivo esclavo) o pase a modo recepción (si ha solicitado leer del dispositivo esclavo). Obviamente el dispositivo esclavo pasará al modo de transmisión/recepción complementario al del maestro.

Los bytes enviados mediante el bus I^2C se transmitirán de forma que el bit más significativo se enviará primero y el bit menos significativo el último. Esto se aplicará tanto para bytes de direcciones como para bytes de datos.

Existen dos tipos especiales de bit empleados para indicar el inicio (bit de **START**) o fin (bit de **STOP**) de una transmisión. El bit de START se indica mediante una transición de nivel alto a nivel bajo de la línea SDA cuando la línea SCL se encuentra a nivel alto. El bit de STOP se indica mediante una transición de nivel bajo a nivel alto mientras el reloj (SCL) se encuentra a nivel alto.



Condiciones START y STOP de una transmisión I^2C [40]

Si el transmisor desea escribir más de un byte sobre el dispositivo esclavo le basta con seguir transmitiendo bytes después de cada bit de ACK recibido del esclavo. Cuando el dispositivo maestro desee finalizar el proceso de escritura le bastará con emitir un bit de STOP después del último ACK recibido.

Si el dispositivo maestro desea leer más de un byte del dispositivo esclavo le basta con emitir un bit de ACK después de cada byte de datos recibido, lo que indica su deseo de continuar con la lectura. Una vez leído el último byte deseado, el maestro contestará con un bit de STOP en lugar de un ACK para finalizar la lectura.

El dispositivo maestro puede, en un momento dado, querer mantener el control de bus para realizar una nueva transferencia de datos (un mensaje combinado). Para ello, en lugar de enviar un bit de STOP enviará un nuevo bit de START (procedimiento de START repetido).

Así pues, I^2C define tres tipos básicos de mensaje, cada uno de los cuales comienza con un código de START y termina con un código de STOP:

- Mensaje simple de escritura desde dispositivo maestro a esclavo
- Mensaje simple de lectura de dispositivo esclavo (enviado desde dispositivo maestro)
- Mensajes combinados, en los que el dispositivo maestro realiza dos o más lecturas y/o escrituras a uno o varios dispositivos esclavos.

En un mensaje combinado, después del envío del primer bit de START y de un mensaje normal, se envían nuevos bits de START (repetidos) no precedidos previamente de un bit de STOP. Esto permite informar a los dispositivos receptores que toda la transferencia de datos anterior y futura (hasta que reciban un byte de STOP) pertenece al mismo mensaje.

Llegados hasta este punto, es importante mencionar que es habitual que cada dispositivo esclavo estrictamente compatible con I^2C responda únicamente a mensajes particulares definidos en su hoja de producto. Esto es debido a que ni I^2C ni TWI definen la semántica de los mensajes, tales como pueden ser el significado de los bytes transmitidos en los tramas. Esta semántica será específica de cada producto o fabricante.

Así pues, en la práctica, la mayor parte de los dispositivos esclavos adoptarán un modelo de petición/respuesta, en donde uno o dos bytes transmitidos después de un comando de escritura, se tratarán como una dirección (por ejemplo un registro de una memoria) o un comando. Esos bytes determinarán cómo se tratarán los siguientes datos emitidos para escritura o cómo habrá de responder el esclavo ante las siguientes lecturas.

3.8.2. Capa Física

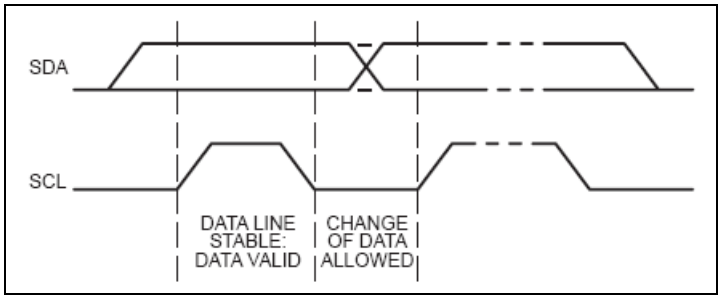
En el bus I^2C, forzar una salida a tierra se considera un cero lógico mientras que dejar la línea SDA flotante a V_{cc} se considera un uno lógico.

Este sistema permite un control de acceso al canal: cuando un dispositivo quiere “escribir” un uno lógico en la línea de datos, ha de dejar el hilo flotando y después comprobar que no sigue a masa; de estar a tierra, significará que ha habido colisión con otro dispositivo maestro que ya estaba utilizando el bus. No obstante, este suceso es poco probable puesto que todo maestro que ha leído un código **START** de la línea de datos SDA, no trata de acceder al canal hasta que escucha un **STOP**. Así pues esto sólo sucederá en el caso de que dos dispositivos empiecen a emitir simultáneamente y se traducirá en que uno de los dos detectará este suceso y dejará de transmitir (pérdida de arbitraje) pero el otro seguirá como si nada hubiese sucedido (y sus datos no perderán integridad).

Este mecanismo de escritura en un canal y posterior comprobación del nivel de éste puede ser empleado en las dos líneas del bus con distintos objetivos:

- En la línea del reloj (SCL) un dispositivo actuando como esclavo puede mantener a nivel bajo el reloj para indicar que necesita tiempo para procesar los datos que ha recibido hasta ahora. De este modo el dispositivo maestro ha de esperar hasta que el esclavo libere la línea para seguir enviándole datos. A este mecanismo de control de flujo para los dispositivos esclavos se le denomina "***Clock Stretching***".
- En la línea de datos (SDA) como sistema de detección de colisión. Aquí este procedimiento (denominado "**arbitraje**") asegura que sólo existe un transmisor de datos en la línea cada vez.

Las transiciones de nivel para los bits en el canal de datos (SDA) han de realizarse siempre mientras la línea de reloj (SCL) esté a nivel bajo (ver figura siguiente). Las transiciones que tengan lugar mientras el reloj se encuentre en nivel alto (V_{cc}) se interpretarán como los códigos de **START** (transición de nivel alto a bajo de SDA) y **STOP** (transición de nivel bajo a alto).



Transiciones de nivel en la línea SDA [40]

3.8.3. I^2C vs. TWI

El Interfaz **TWI** (*Two Wire Interface*) es esencialmente el mismo bus I^2C implementado en diversos procesadores *system-on-chip* diseñados por **Atmel** y otras compañías. Así pues es compatible con el diseño de referencia y capa física I^2C descritos en los apartados anteriores.

Muchos fabricantes emplean el nombre **TWI** en lugar de I^2C debido a cuestiones de licencias de copyright. A día de hoy, el bus I^2C se ha convertido en un estándar de facto y ya no es necesario adquirir licencias para emplearlo.

A efectos de este proyecto, el ND07 integra en su placa la plataforma MC13213 de *Freescale*, que dispone de un controlador serie I^2C, y la EEPROM de Atmel AT24C512, cuya hoja de producto especifica que se comunica mediante un bus TWI.

Las pequeñas diferencias que podrían existir entre I^2C y TWI se darían sobre todo en lo relativo a las velocidades máximas que es capaz de alcanzar el bus I^2C en sus últimas revisiones (que aún no incorpora o ratifica TWI). Adicionalmente, aquellos dispositivos incapaces de adoptar

todas las especificaciones del bus I^2C se considerarán a sí mismos como adoptantes del bus TWI en lugar del I^2C.

No obstante la EEPROM AT24C512 otorga una velocidad máxima de 400 Kbit/s (mediante la alimentación suministrada en placa) de modo que a efectos de este proyecto TWI e I^2C representarán el mismo concepto. Así pues, a lo largo de este documento se empleará siempre la nomenclatura I^2C en lugar de TWI para referirse al bus por ser de entre los dos el bus más extendido y conocido en el mercado (pero se referirán siempre al mismo tipo de bus).

3.8.4. I^2C en el AT24C512

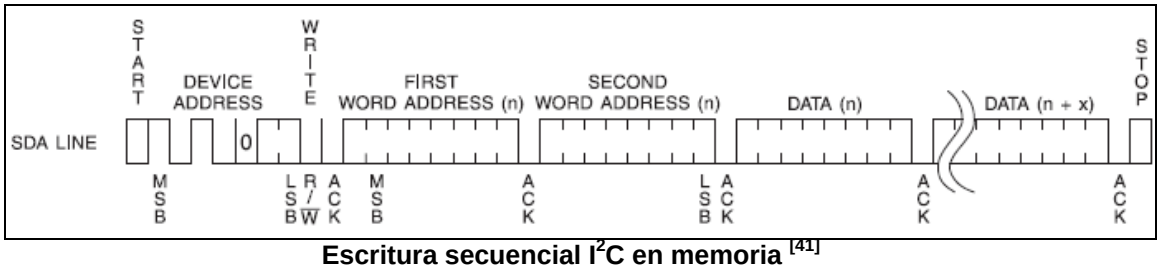
El bus empleado para comunicar el microcontrolador MC13213 de *Freescale* y la EEPROM AT24C512 de Atmel es compatible con la definición de bus I^2C (o TWI) estudiado anteriormente. Sin embargo la EEPROM, como ya se ha sugerido con antelación para muchos dispositivos esclavos, emplea una semántica específica para modificar o acceder mediante el bus I^2C a los datos que almacena.

Así pues, el microcontrolador deberá seguir los siguientes pasos para comunicarse con la EEPROM dependiendo de si se desea leer o escribir sobre ésta:

3.8.4.1. Escritura sobre la EEPROM AT24C512

- El microcontrolador (que actuará siempre como dispositivo maestro) comienza la comunicación enviando un bit START. Esto alertará a la EEPROM esclava, poniéndola a la espera de una transacción.
- El maestro se dirige al dispositivo con el que quiere comunicarse (la EEPROM), enviando un byte que contiene los siete bits que componen su dirección (en el caso del ND07, la EEPROM tiene la dirección 1010000. Ver apartado 3.5), y el octavo bit de menor peso que se corresponde con la operación deseada: en este caso escritura (bit a 0).
- La dirección enviada es comparada por la EEPROM con su propia dirección y, si ambas coinciden, la memoria se considera direccionada como dispositivo esclavo-receptor (ya que el bit R/W indicaba escritura).
- La EEPROM responde enviando un bit de ACK que le indica al microcontrolador que ésta reconoce la solicitud y que está en condiciones de comunicarse.
- A continuación el microcontrolador enviará la posición de memoria de la EEPROM sobre la que quiere escribir (puesto que es una memoria de 64KB esta dirección será de 2 bytes).

- El dispositivo esclavo responderá con un bit de ACK después de cada byte de posición de memoria correctamente recibido.
- Ahora el microcontrolador puede empezar a escribir bytes de datos sobre la posición de memoria de la EEPROM auxiliar y sus posiciones consecutivas. Todos los bytes de datos deben constar de 8 bits y el número de bytes que pueden ser enviados en una misma transmisión no debe superar el máximo especificado por el dispositivo destino. En el caso de la EEPROM AT24C512 este máximo se sitúa en 128 bytes. Por encima de ese número los datos se sobrescribirán sobre sí mismos.
- Cada byte escrito por el dispositivo maestro debe ser obligatoriamente reconocido por un bit de ACK por la EEPROM.
- Cuando la comunicación finaliza, el maestro transmite un bit de STOP para dejar libre el bus.
- Después del bit de STOP, es obligatorio para el bus estar inactivo durante unos microsegundos.

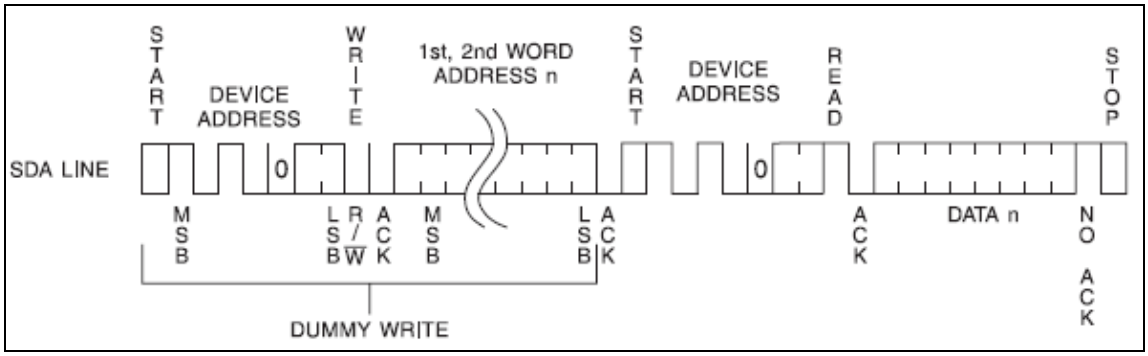


Escritura secuencial I^2C en memoria [41]

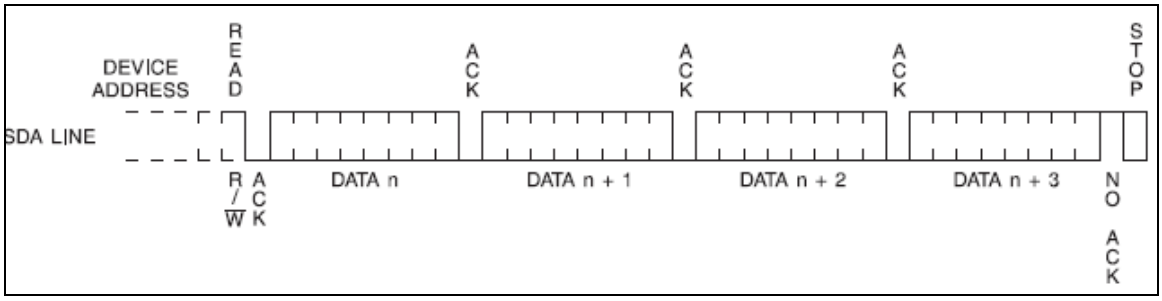
3.8.4.2. Lectura de la EEPROM AT24C512

- Para leer posiciones de memoria almacenadas en la EEPROM AT24C512 es preciso que el microcontrolador realice previamente una petición de escritura sobre el bus indicando la posición de memoria sobre la que se desea efectuar la adquisición de datos. Así pues, para realizar lecturas de la EEPROM es imprescindible realizar previamente los seis primeros pasos descritos en el apartado anterior correspondientes a la escritura de datos sobre la memoria.
- Una vez informada la EEPROM acerca de la posición de su memoria sobre la que se desea trabajar, ésta responderá con un bit de ACK y esperará que los datos que se envíen a continuación sirvan para sobrescribir la posición de memoria apuntada.
- Es en este momento en que el microcontrolador vuelve a emitir un bit de START (START repetido) indicando que se trata de un mensaje compuesto y que pretende leer en lugar de escribir sobre la memoria.
- Para ello, después del bit de START repetido, el microcontrolador vuelve a enviar a la EEPROM un byte con su dirección y con el bit R/W indicando modo lectura (bit a 1).

- La EEPROM responde enviando un bit de ACK que le indica al microcontrolador que ésta reconoce la solicitud y que está en condiciones de enviar las lecturas solicitadas.
- A continuación la EEPROM procederá a enviar bytes sucesivamente partiendo de la posición de memoria fijada previamente al principio de la comunicación.
- El microcontrolador asentirá cada byte de datos enviando un bit ACK hasta que no desee leer más (esta lectura puede durar indefinidamente), momento en que no enviará el bit de ACK (o lo que es lo mismo enviará un bit de NACK) y después transmitirá un bit de STOP para finalizar la comunicación y dejar libre el bus.
- Después del bit de STOP, es obligatorio para el bus estar inactivo durante unos microsegundos.



Lectura I²C de un único byte concreto en memoria [41]



Lectura I²C secuencial de bytes en memoria [41]

Así pues, concluida la introducción acerca del funcionamiento del bus I²C, nos encontraremos en el escenario adecuado para afrontar el desarrollo del interfaz de comunicación entre el ND07 y su memoria EEPROM a través del módulo I²C de la plataforma MC13213.

Una vez concluido dicho interfaz, se empleará para el almacenaje y posterior lectura de imágenes de *firmware* destinadas a actualizar la memoria del ND07.

3.9. Estándar RS232

El estándar RS-232 (*Recommended Standard 232*) define una interfaz para el intercambio serie de datos binarios entre un DTE (equipo terminal de datos) y un DCE (equipo de comunicación de datos).

Aunque fue publicado por la EIA (*Electronic Industries Alliance*) en los años sesenta, el estándar ha sufrido pocas modificaciones y aún se emplea habitualmente en los puertos serie de ordenadores. La revisión más extendida es la RS-232C (1969) y la última que se ha publicado es la RS-232E.

Las recomendaciones publicadas por la ITU (*International Telecommunication Unit*) **ITU-T V.24** junto con la **ITU-T V.28** son equivalentes al RS232.

El estándar define los siguientes parámetros:

- Características eléctricas de las señales transmitidas: niveles de voltaje, velocidad, temporizaciones y forma de las señales, máxima capacitancia de la línea, etcétera.
- Características mecánicas del interfaz: conectores e identificación de pines.
- Función de cada circuito del conector.

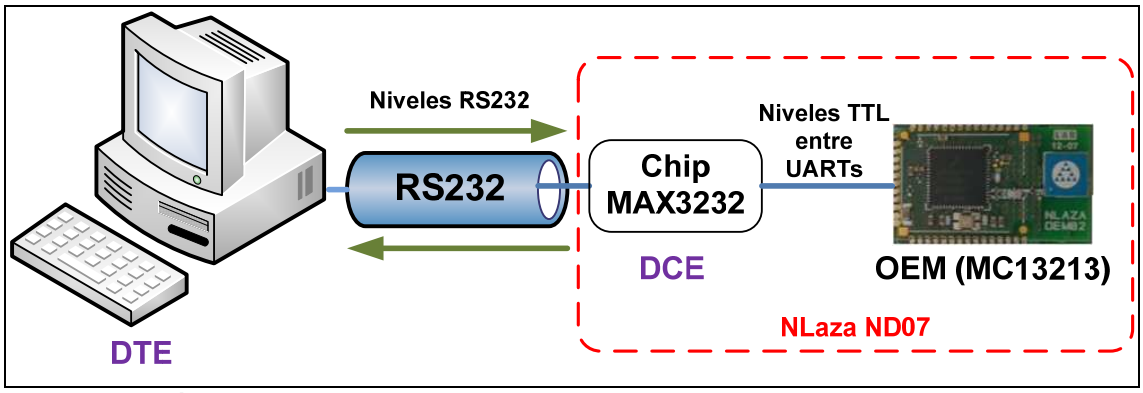
Sin embargo, deja abiertas cuestiones como:

- Codificación de los caracteres (ASCII, EBCDIC, etc.)
- Formato de tramas en el flujo de datos: bits por carácter, bits de inicio/fin (cuando se realizan comunicaciones asíncronas), bits de paridad, etc.
- Protocolos para la detección de errores o algoritmos para la compresión de datos
- Velocidades de transmisión de bits (que actualmente pueden llegar a superar los 115,200 bps)

Habitualmente parámetros como la codificación de los caracteres o la velocidad de transmisión son controlados por el hardware del puerto serie, que suele ser un circuito integrado que implementa una UART capaz de convertir un determinado tráfico de datos en paralelo a un flujo serie asíncrono de bits.

Por otro lado, una etapa especial en los extremos de la comunicación se encargará de adecuar los niveles lógicos de voltaje que emplea la UART a los niveles de señal que emplea RS232.

A continuación se presenta un esquema de la comunicación serie RS232 que tendrá lugar entre el PC y el ND07, a través del chip MAX3232 que integra. A su vez, el chip MC13213 se comunicará con el integrado MAX3232 a través de su UART (mediante señales TTL).



Comunicación RS232 entre un PC (DTE) y el chip MAX3232 (DCE) integrado en el ND07

RS232 soporta la transmisión síncrona y asíncrona de datos y, puesto que el estándar define diferentes circuitos físicos unidireccionales para transmitir bits desde el DTE hacia el DCE y viceversa, el interfaz provee de una comunicación *full-duplex*.

A continuación se describirán los parámetros de la comunicación que el estándar RS232 estipula.

3.9.1. Niveles de Voltaje

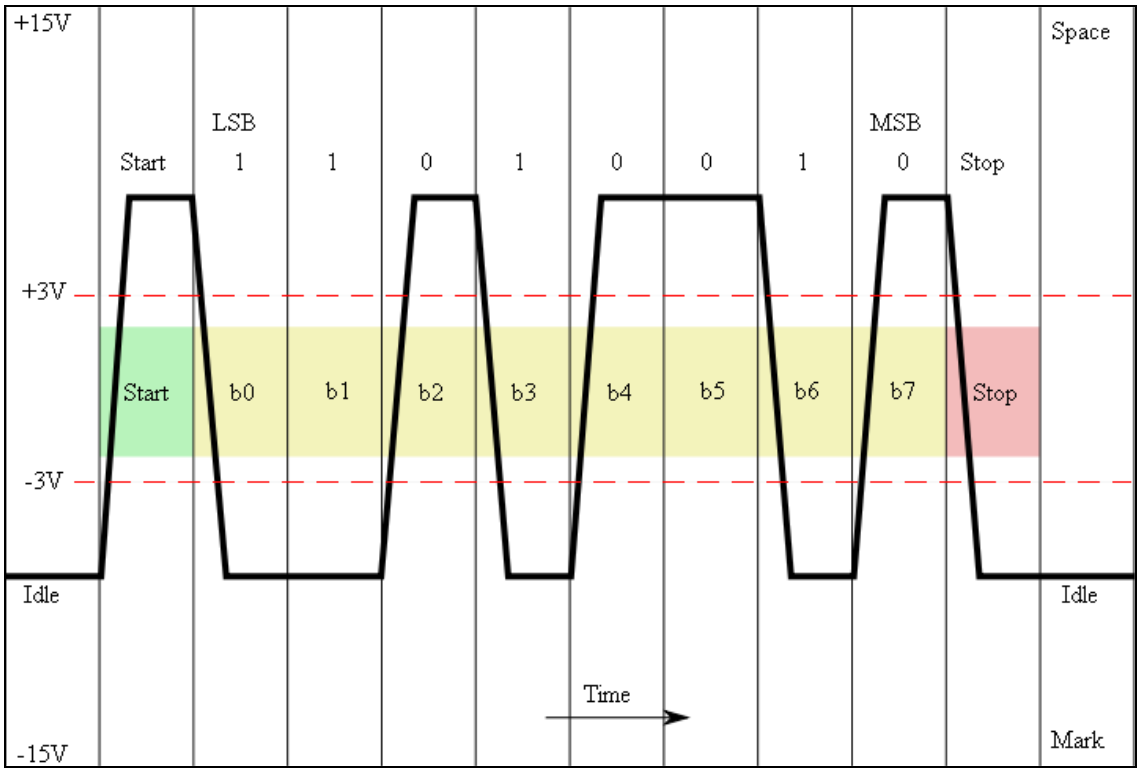
El estándar RS232 estipula una serie de parámetros relacionados con los niveles de voltaje y con los circuitos de adaptación de las señales que se resumirán a continuación.

- Un uno binario se denominará “marca” (*mark*) y se representará por un voltaje de -3 a -15 voltios.
- Un cero binario se denominará “espacio” (*space*) y se representará por un voltaje de +3 a +15 voltios.
- Cualquier voltaje entre -3 y +3 voltios se considerará inválido.
- Las señales de control serán activas “a nivel bajo”. Esto quiere decir en este caso que se activarán mediante señales con un voltaje de +3 a +15 voltios.
- Mientras no se envíen datos la señal se debe mantener en estado de “marca” (un uno lógico, conocido también como *RS-232 idle state*)
- Se soportará un voltaje máximo de 25 voltios en circuito abierto.

Debido a que los voltajes empleados son muy superiores a los niveles lógicos empleados habitualmente por circuitos integrados, suele ser necesario emplear circuitos especiales de

adaptación (*drivers*) que realizarán dicha transformación de niveles. Adicionalmente, se encargarán de suministrar la corriente necesaria a los circuitos para asegurar la correcta transmisión de señales y protegerán al sistema frente a cortocircuitos y frente a efectos transitorios del interfaz.

En la imagen siguiente puede apreciarse un ejemplo de transmisión de un byte a través de una comunicación RS232 que cumple los requisitos anteriormente mencionados:



Ejemplo de transmisión del carácter "K" (0x4B) con 8 bits de Datos, 1 de Parada y 1 de *Start*

Una limitación del estándar es que, debido a que ambos extremos de la comunicación dependen de compartir una tierra común, pueden llegar a darse problemas de transmisión cuando el DTE y el DCE se encuentren muy alejados, e incluso generarse bucles de masa (corrientes indeseadas en un conductor entre dos puntos que se presuponen al mismo voltaje).

3.9.2. Conectores y Señales

Aunque el estándar define dos tipos de dispositivos (el DTE y el DCE) para poder distinguir el sentido de las señales de cada circuito, no requiere el uso de un conector determinado (pero recomienda el DB-25).

Por lo general (aunque existen excepciones), los equipos clasificados como DTEs (terminales, ordenadores, etc.) dispondrán de conectores machos con las funciones de los pines asociadas a dicha clasificación. Por otro lado, los equipos clasificados como DCEs (módems, etc.) dispondrán de conectores hembras y su correspondiente identificación de pines.

Aunque el estándar define 20 líneas de señal diferentes, los sistemas más habituales sólo emplean un grupo reducido de éstas, de modo que se han popularizado muchos conectores diferentes como son: DB-25, DE-9, EIA/TIA 561, Yost, etc.
A efectos del escenario de aplicación de este proyecto, se espera que el ordenador conectado con el ND07 adopte el rol de DTE y disponga de un puerto serie con conector DE-9 macho para acoplarse al ND07 mediante un cable RS232 estándar (ya que el ND07 actuará siempre como DCE y dispone de un conector DE-9 hembra).

De este modo, las señales implementadas más habitualmente en los conectores serán las siguientes:

Pin (DE-9)	**Señal**	**Etiqueta**	**Descripción**
1	*Carrier Detect*	DCD	Empleada por el DCE para indicar que se ha establecido una conexión con un equipo remoto
2	*Receive Data*	RxD	Datos del DCE al DTE
3	*Transmitted Data*	TxD	Datos del DTE al DCE
4	*Data Terminal Ready*	DTR	Empleada por el DTE para indicar que está listo para conectarse. Esta señal se emplea para sacar al DCE de estados de hibernación.
5	*Signal Ground*	G	Tierra del circuito
6	*Data Set Ready*	DSR	Empleada por el DCE para indicar que está alimentado y listo para recibir conexiones y transmisiones de datos
7	*Request To Send*	RTS	Empleada por el DTE para notificar al DCE su deseo de transmitirle datos
8	*Clear To Send*	CTS	Empleada por el DCE para asentir el RTS y permitir al DTE comenzar a transmitirle datos
9	*Ring Indicator*	RI	Empleada por el DCE para indicar al DTE que tiene una llamada

Señales / pines de un conector serie RS232 DE-9

La revisión RS232-E del estándar contempla un funcionamiento especial (simétrico) para los pines CTS y RTS denominado "*RTS/CTS handshaking*", en el que se define una nueva señal,

Ready to Receive (RTR), sobre el circuito RTS de forma que tanto el DTE como el DCE pueden permitir o denegar la recepción de datos.

3.9.3. Configuración

Así pues, se puede concluir que el estándar RS232 especifica una serie de parámetros (la capa física) de la comunicación entre un DTE y DCE y recomienda el uso de una serie de conectores y protocolos sobre sus pines.

Sin embargo, queda claro que el resto de parámetros de la comunicación: empleo de señal síncrona o asíncrona (y con ella, si procede, el número de bits de inicio y parada), velocidad de transmisión, codificación de caracteres (número de bits de datos y paridad), control de flujo (hardware, mediante las líneas CTS/RTS o software, mediante los caracteres especiales *Xon* y *Xoff*), etcétera.

A lo largo de este proyecto se empleará siempre una comunicación serie RS232 entre un PC (DTE) y un ND07 (DCE) asíncrona, de 19200 baudios, con 8 bits de datos, un bit de parada y sin bit de paridad. Adicionalmente será necesario configurar el uso de control de flujo hardware sin *CTS/RTS handshaking.*

Capítulo 4
Bootloader y Aplicaciones Auxiliares

4.1. Bootloader

En este capítulo se tratará principalmente de explicar el trabajo desarrollado para lograr programar el **bootloader**, objetivo último de este proyecto, así como la aplicación ZigBee "**cargador RS232**" que permitió depurar y validar el correcto funcionamiento de éste.

De este modo, el primer paso a dar será definir más profundamente el concepto de bootloader y su alcance, facilitando así la comprensión de los pasos efectuados para completar su diseño (los cuales se desarrollarán a lo largo del resto de aparatados de este capítulo).

A continuación, será imprescindible dedicar varios subapartados a describir todos aquellos componentes hardware del MC13213, módulos de las librerías ZigBee y características del *firmware* en general cuyo funcionamiento afecta directamente al bootloader de un modo u otro. De cada uno de estos apartados se concluirán decisiones de diseño y especificaciones de configuración (que se programarán en el código definitivo) que serán imprescindibles para desarrollar un bootloader flexible que sea instalable en aplicaciones ZigBee.

Inmediatamente después, se estudiará el formato de datos empleado por las imágenes de *firmware* (registros S19) y basándonos en él se diseñará el formato óptimo que se empleará para almacenar las mismas en la memoria auxiliar EEPROM mientras dure el proceso de una actualización (o distribución de imágenes). Una vez más, este diseño condicionó el modo en que se implementaron las aplicaciones finales del bootloader y del cargador RS232.

Posteriormente, estudiados ya todos los elementos relacionados con el proyecto, se procederá por fin a describir la arquitectura y el funcionamiento de los bloques que conforman la aplicación desarrollada, demostrando el cumplimiento de las especificaciones requeridas como bootloader, objetivo último de este proyecto.

Dentro de este subapartado será necesario incluir también el trabajo realizado que ha sido necesario para otorgar a la aplicación del bootloader de las propiedades de flexibilidad, robustez e independencia de la aplicación principal ZigBee que eran requisitos imprescindibles de diseño. Fruto de este trabajo se desprenden los subapartados de validación de la actualización, configuración del compilador y del linkador, traducción del código a ensamblador, etcétera.

Para concluir el apartado, se han incluido también las restricciones de diseño y trabajo adicional que el servicio técnico de *Freescale* y la publicación de actualizaciones de librerías han representado para alcanzar el objetivo final buscado.

.

4.1.1. Introducción: Conceptos Básicos

Un cargador de arranque (*boot loader* en inglés) es un programa sencillo diseñado exclusivamente para preparar todo lo que necesita el sistema para funcionar pero que no suele disponer de todas las funcionalidades de un sistema operativo. Normalmente se utilizan los cargadores de arranque multietapa, en los que varios programas pequeños se suman los unos a los otros, hasta que el último de ellos carga el sistema operativo.

En los ordenadores modernos, el proceso de arranque comienza con la CPU ejecutando los programas contenidos en la memoria ROM en una dirección predefinida (se configura la CPU para ejecutar este programa, sin ayuda externa, al encender el ordenador). El último de estos programas se encargará de cargar el sistema operativo propiamente dicho y de transferirle el control.

El proceso de arranque se considera completo cuando el ordenador está preparado para contestar a los requerimientos del exterior. El típico ordenador moderno arranca en, aproximadamente, un minuto (del cual, 15 segundos son empleados por los cargadores de arranque preliminares y, el resto, por el cargador del sistema operativo), mientras que los grandes servidores pueden necesitar varios minutos para arrancar y comenzar todos los servicios.

En cambio, la mayoría de los sistemas empotrados deben arrancar casi instantáneamente ya que, por ejemplo, esperar un minuto para poder ver la TV se considera inaceptable. Por ello, tienen el sistema operativo grabado en la ROM o memoria Flash, gracias a lo cual puede ejecutarse de forma casi instantánea. Este es el caso de los dispositivos ZigBee con los que se ha desarrollado este proyecto.

Sin embargo, en los entornos de redes de sensores ZigBee del mercado, es bastante habitual encontrarse el término bootloader en las hojas de producto de librerías y de dispositivos referenciando a la capacidad de que disponen de actualizar su *firmware* en tiempo de ejecución (principalmente de forma inalámbrica). De este modo, es común que se confunda el proceso de arranque de un dispositivo con el conjunto formado por el protocolo de carga remota de dispositivo (OTAP [18]) y el Bootloader.

Las librerías ZigBee que *Freescale* suministra para la plataforma MC13213 (integrada dentro del dispositivo ZigBee NLaza ND07, sobre el que se desarrolla este proyecto) proveen de un cargador de arranque básico que carece de la funcionalidad de actualización del *firmware* del equipo. Adicionalmente, este cargador de arranque (diseñado para funcionar sobre placas de desarrollo de *Freescale*) ha de ser modificado para contemplar todas las funcionalidades adicionales que implica la integración del microcontrolador MC13213 junto con los diversos periféricos del dispositivo NLaza ND07.

El objetivo final de este proyecto será modificar el cargador de arranque ya existente, introduciéndole una etapa adicional capaz de detectar si se encuentra almacenado nuevo *firmware* listo para sustituir al actual y, si es así, realizar dicho reemplazo. Será esta nueva etapa pues, la que de ahora en adelante denominaremos **Bootloader**. El modo en que el nuevo *firmware* es transmitido hasta el dispositivo a actualizar y la forma en que se comprueba su integridad para luego almacenarse en la memoria auxiliar quedan pues fuera del alcance de este proyecto.

En cambio, el formato del *firmware* almacenado en la memoria auxiliar sí será otro objetivo de diseño de este proyecto ya que su definición y optimización son fundamentales para el entendimiento entre la aplicación principal, encargada de recibir y almacenar la nueva imagen, y el bootloader, encargado de procesarla para actualizar el *firmware* del equipo.

De este modo, el *firmware* de una aplicación sencilla ZigBee dotada de la capacidad de actualización de su *firmware* seguirá un esquema similar al de la figura siguiente:

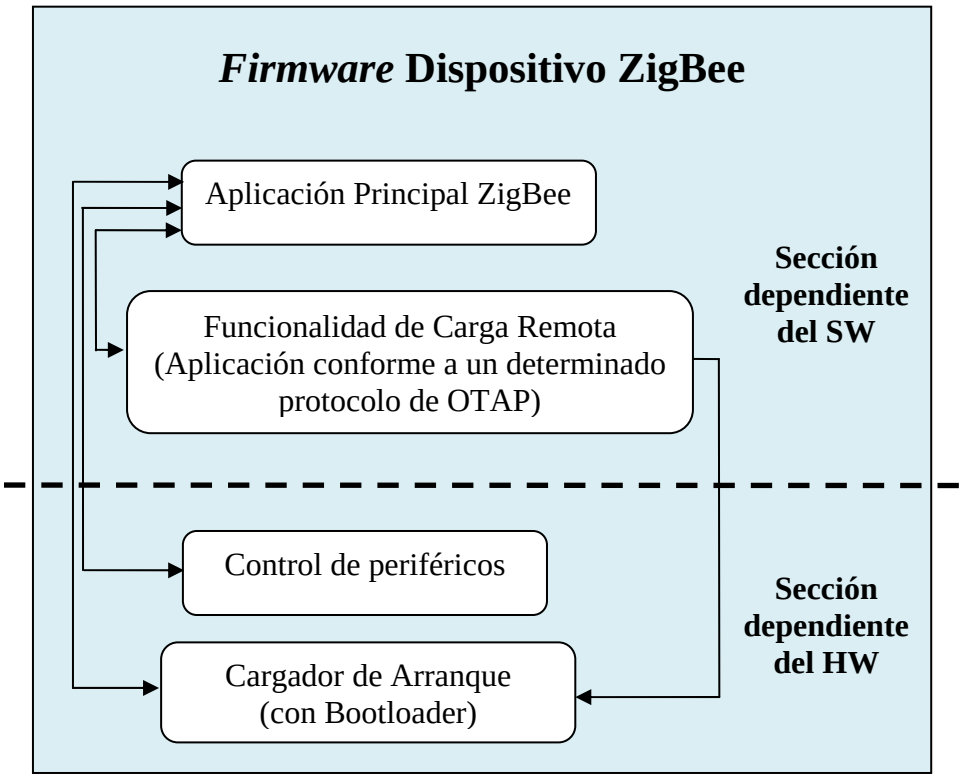


Diagrama funcional del *firmware* de un dispositivo ZigBee con bootloader

Como puede observarse, el cargador de arranque será distinto según sea el hardware sobre el que se programe. Por otro lado, el protocolo que la aplicación principal emplee para recibir el nuevo *firmware* podrá ser desarrollado de forma casi transparente a la electrónica concreta del dispositivo a actualizar.

Será esta propiedad teórica del bootloader, su independencia de la aplicación ZigBee principal y del protocolo de adquisición de imágenes de *firmware*, uno de los principales objetivos de diseño que se buscarán a la hora de programarlo. De este modo, la instalación del bootloader en un dispositivo ZigBee permitirá a la aplicación principal actualizar de forma flexible su *firmware*, incluso si ésta modifica secciones del cargador de arranque o el protocolo de adquisición de imágenes (o incluso si éste deja de ser inalámbrico).

Es necesario tener en cuenta que el proceso de actualización del *firmware* de un dispositivo plantea un claro inconveniente: la aplicación no puede sobrescribirse en tiempo de ejecución. Esto quiere decir que un dispositivo no puede actualizar su memoria paulatinamente según va recibiendo secciones del nuevo *firmware* ya que la sobrescritura de sectores de memoria al azar podría provocar que la aplicación se corrompiese (actualizándose a medias) y el equipo quedase inutilizable.

Para solucionar este problema, la aplicación principal deberá encargarse (cuando proceda) de recibir, comprobar la integridad y almacenar el nuevo *firmware* completo para después reiniciar controladamente el equipo (reset *ordenado*). Será durante su siguiente secuencia de arranque (antes de reincorporarse a la red ZigBee), cuando el bootloader tomará el control del dispositivo y actualizará completamente su *firmware* (a excepción de la pequeña sección de memoria Flash ocupada por el propio bootloader).

Esta actualización será inicializada por el bootloader ante una señal emitida por la aplicación principal (mediante la activación de un *flag* en la memoria auxiliar y el *reseteo* del equipo) siempre que se haya completado con éxito la recepción y el almacenamiento del nuevo *firmware* que pretende sustituir al actual. Dicha notificación desde la aplicación principal hacia el bootloader mediante un *flag* en memoria será lo más parecido a una comunicación que se dará entre ambas secciones de código.

Aunque el bootloader será siempre una funcionalidad muy interesante a instalar en todo dispositivo ZigBee, habitualmente se hará uso de él pocas veces a lo largo de su vida útil. Es por esta razón que es vital que los recursos que consuma (en términos de memoria ocupada) sean lo más reducidos que sea posible. Éste será pues el último objetivo claro de diseño a la hora de programar el bootloader.

4.1.2. Elementos Relacionados con el Bootloader

A lo largo de los subapartados siguientes se explicarán todos aquellos componentes hardware del MC13213, módulos de las librerías ZigBee y características del *firmware* en general cuyo funcionamiento afecta al bootloader de un modo u otro.

De este modo, fruto del estudio de cada componente en particular se extrae al final de cada subapartado una serie de conclusiones:

- Decisiones de diseño del bootloader para asegurar el funcionamiento buscado.
- Configuraciones que es necesario programar en el módulo bajo estudio para asegurar la compatibilidad con el bootloader.
- Recomendaciones de diseño de aplicaciones ZigBee y configuración de módulos del MC13213 para desarrolladores que deseen instalar el bootloader dentro del código programado.

Las conclusiones obtenidas en estos subapartados determinarán no sólo el modo en que se diseñará la aplicación del bootloader sino también los requisitos que habrán de cumplirse en el resto de módulos para que el funcionamiento de éste sea el óptimo.

4.1.2.1. Módulos de la Secuencia de Arranque

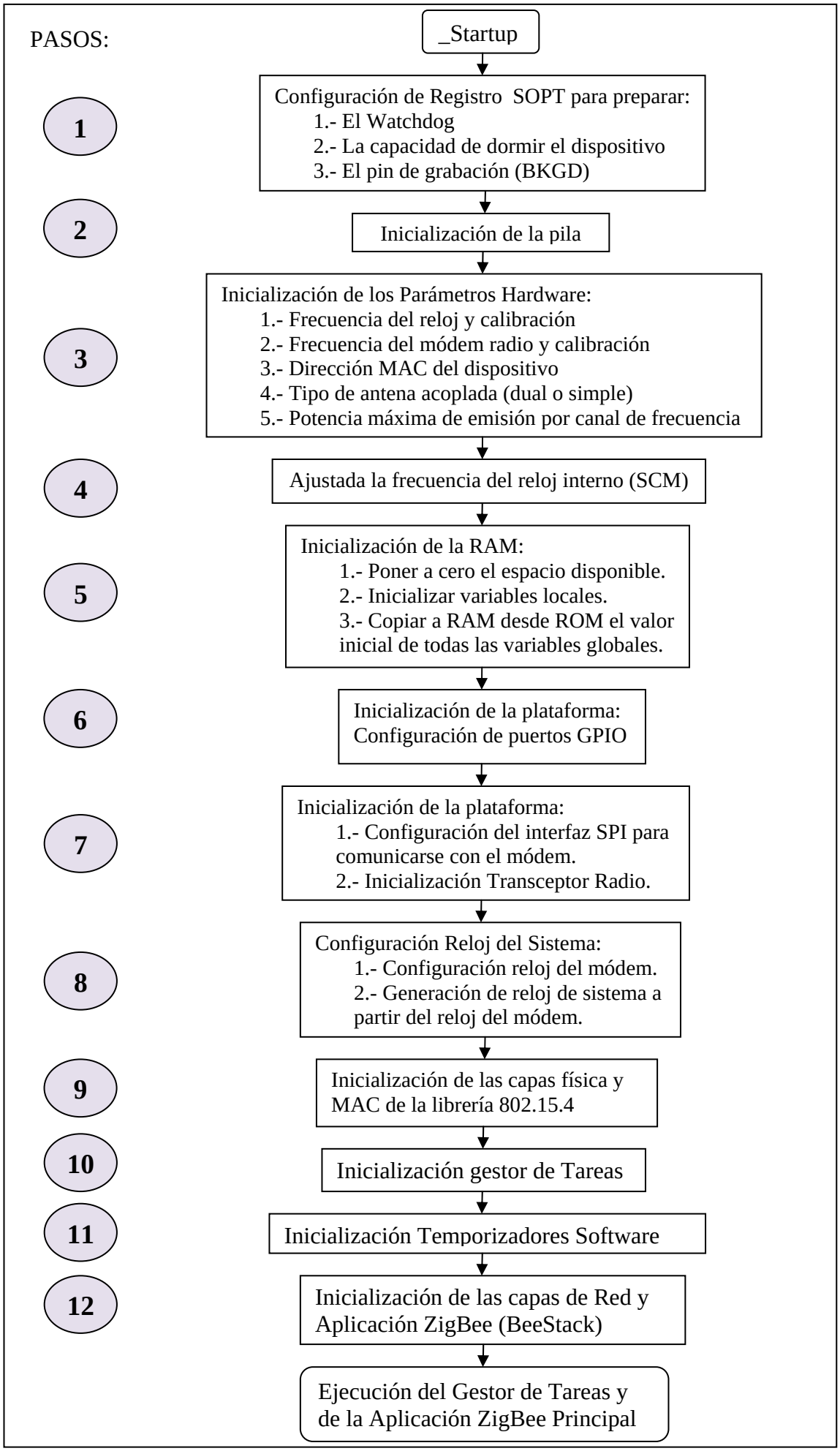
Como ya se ha mencionado, las librerías ZigBee y 802.15.4 suministradas por *Freescale* para la plataforma MC13213 incluyen códigos de aplicaciones de ejemplo diseñados para acelerar el desarrollo y el *time-to-market* de los dispositivos que integren este chip.

Puesto que el dispositivo NLaza ND07 integra en su placa la plataforma MC13213, lo más lógico será comenzar a desarrollar las aplicaciones ZigBee para este equipo a partir de los ejemplos suministrados por *Freescale (*con su correspondiente secuencia de arranque de ejemplo). Es por esto que a lo largo de todo el proyecto se desarrollará un bootloader que se integrará dentro de dicha secuencia de arranque de ejemplo.

Aunque no se explicará concienzudamente cada uno de los elementos implicados en el arranque del dispositivo por no ser éste objetivo del proyecto, ha sido imprescindible estudiar cada uno de ellos por separado para adaptarlos correctamente al hardware concreto del ND07 y para evaluar su posible influencia sobre el correcto funcionamiento del bootloader. Esto se debe a que la configuración por defecto de los códigos de ejemplo está preparada para funcionar con placas de desarrollo de *Freescale*.

No obstante, se han realizado pocas modificaciones de la secuencia de arranque original ya que ésta, a excepción de la configuración del módulo I^2C y algunos puertos y registros, era considerablemente compatible.

Así pues la secuencia de arranque del ND07, antes de la inclusión del bootloader dentro de ella (decisión que se discutirá en el apartado 4.1.4.5.2), seguirá los siguientes pasos:



Pasos de la secuencia de arranque de una aplicación ZigBee

Como puede deducirse del esquema anterior, la secuencia de arranque del ND07 comprende principalmente la inicialización de diversos módulos del MC13123 mediante la configuración de registros concretos de la memoria Flash.

El orden de los elementos dentro de la secuencia no es arbitrario y se debe principalmente a los siguientes factores:

- Existen módulos, como el encargado de la protección de sectores de la memoria o el watchdog, que requieren ser configurados en las primeras etapas del encendido del dispositivo.
- La inicialización de la memoria volátil, compuesta por la pila (*memory stack*) y el resto de la RAM, es un paso que ha de ser dado previo a la ejecución de cualquier aplicación que haga uso de variables y constantes.
- Cuanto más tarde se inicialice un módulo más expuesto estará a sufrir posibles interrupciones por parte de otros módulos ya inicializados. Es por ello que las operaciones críticas tenderán a inicializarse lo antes posible dentro de la secuencia de arranque, haciéndolas razonablemente independientes del resto de la aplicación (esto se aplicará también al bootloader).
- La configuración de los puertos de entrada/salida (*GPIO*) no es prioritaria pero deberá de realizarse sin duda antes que la inicialización de los módulos que pretendan hacer uso de ellos.
- La plataforma MC13123 puede generar su reloj de referencia empleando para ello diversas fuentes, sin embargo la mayor parte de ellas requieren de un periodo de estabilización y sincronización previo a su uso. Es por ello que para poder arrancar el equipo, la plataforma emplea inicialmente como reloj del sistema una señal interna de baja precisión y frecuencia. De este modo sólo aquellos módulos que no requieran de una precisión muy rigurosa o alta velocidad podrán inicializarse en las primeras fases del arranque.
- Lo más habitual será que, para aumentar la frecuencia y precisión del sistema, a lo largo del arranque se vaya preparando un reloj de referencia alternativo (empleando para ello una señal externa como un cristal o la salida del modem radio) y se comience a utilizar poco antes de otorgarle el control a la aplicación principal. De este modo, todos los módulos que requieran de velocidad de proceso y de un sistema estable se ejecutarán en las últimas fases del arranque (o ya después de terminar éste).
- Aunque el módulo SPI de la plataforma MC13213 se utiliza fundamentalmente para comunicarse con el módem radio del chip y esto no debería ser un factor prioritario, se inicializará relativamente pronto dentro de la secuencia de arranque. Esto se debe a que el módem radio 802.15.4 proveerá de una señal de salida que habitualmente se empleará como referencia para generar un reloj de sistema rápido y preciso (32MHz).

Este reloj será el elegido siempre por defecto en todas las aplicaciones ZigBee del ND07.

- Sólo en una entorno con un reloj de sistema preciso y rápido podrá ejecutarse el gestor de tareas (que se basa principalmente en gestionar el control de diversas secciones de código basándose en temporizadores software y hardware) y los módulos y capas (MAC, Red y Aplicación) que pretenden funcionar sobre éste.

Como conclusión, parece obvio que la secuencia de pasos de arranque de la plataforma MC13213 es bastante coherente tal y como se presenta en las aplicaciones de ejemplo suministradas por *Freescale* y ha requerido pues de pocas modificaciones para adaptarla al caso concreto del hardware del ND07 (aunque sí ha sido necesario un considerable estudio de los módulos implicados).

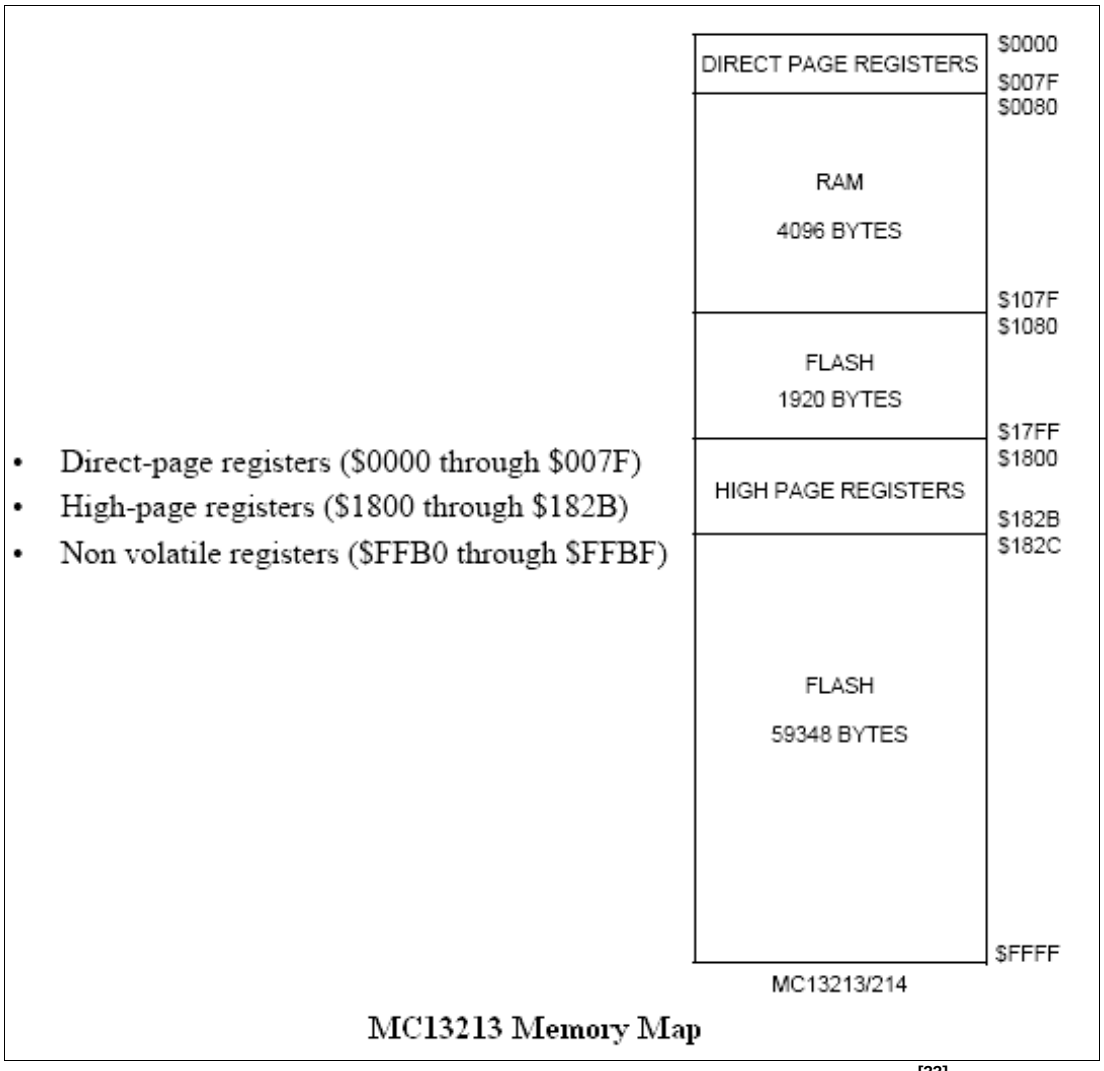
De este modo, la única modificación significativa que se realizará sobre la secuencia de pasos del arranque será la inclusión dentro de ésta del bootloader desarrollado a lo largo del proyecto.

Las razones que justificarán la elección del que será su emplazamiento definitivo dentro de la secuencia de arranque se discutirán en el apartado 4.1.4.5.2, cuando se hayan desarrollado previamente todos los módulos implicados en el proyecto.

4.1.2.2. Memoria del MC13213

El chip MC13213 de *Freescale* integrado en las placas ND07 sobre las que se desarrolla este proyecto, dispone de 64 KB de memoria direccionables repartida entre RAM, Flash y registros de entrada/salida o control/estado.

La figura siguiente muestra los bloques mencionados y su direccionamiento.



Mapa de direcciones de memoria de la plataforma MC13213 [33]

La memoria del MC13213 está compuesta por 128 páginas de 512 bytes que pueden agruparse en los siguientes bloques:

1.- Registros de Acceso Directo (primer cuarto de la página número 0). De este modo los 128 primeros bytes de la memoria corresponderán a los registros de acceso más frecuente (puertos, temporizadores, controladores serie, I^2C y SPI, frecuencia de reloj, etc.) que al residir en la parte más baja del mapa posibilitan direccionamiento directo en ensamblador (direccionamiento más eficiente que reducirá el tamaño total del código generado).

2.- Memoria RAM, 4KB emplazados entre el segundo cuarto de la página número 0 y el primer cuarto de la página número 8 (ambos inclusive).

3.- Bloque Bajo de Memoria ROM. Pequeño segmento de Flash que ocupa desde el segundo cuarto de la página número 8 hasta el final de la página número 11.

4.- Registros de Paginación Alta. Bloque de memoria reservado para aquellos registros de uso muy poco frecuente. Sólo ocupan los primeros 44 bytes de la página número 12 de la memoria del MC13213.

5.- Bloque Alto de Memoria ROM. Esta sección de la memoria está compuesta por las 116 páginas restantes (desde poco después del principio de la página 12 hasta el final de la página 127). Es en esta sección donde se almacenará la mayor parte del código de aplicación.

Al final de la última página (número 127) se encuentran los registros no-volátiles de la Flash y los vectores de interrupción y reset (incluyendo el vector de interrupción de la I^2C necesario para mantener la comunicación con la memoria auxiliar EEPROM externa). Es por ello que este sector será el último en borrarse y regrabarse (como ya se explicará en mayor profundidad más adelante).

Las características genéricas más destacables de la memoria del MC13213 a efectos del bootloader y la actualización de *firmware* son las siguientes:

- Limitación de un máximo de 100.000 ciclos de borrado y reprogramación de la Flash.
- Capacidad de protección de sectores de memoria Flash contra borrado/regrabado accidental o intencionado. Posibilidad de redireccionar los vectores de interrupción a posiciones de memoria no protegidas.
- Seguridad para acceso no autorizado a determinados sectores de memoria.
- No es necesaria una alimentación especial para el borrado y regrabado de la memoria. Basta con la misma que se utiliza para alimentar el circuito durante sus ciclos normales de funcionamiento.
- Interfaz sencillo para la programación de ciclos de borrado y actualización de bloques de memoria Flash.

No obstante la memoria Flash de la plataforma MC13213 presenta una serie de restricciones generales que han de respetarse obligatoriamente:

- La frecuencia del reloj del módulo Flash registro ha de configurarse para que presente un valor de entre 150 KHz a 200 KHz a partir de la frecuencia de reloj del bus del MC13213. Esta operación ha de realizarse siempre antes de la cualquier solicitud de ejecución de comandos al controlador Flash y sólo puede efectuarse una vez después de cada reseteo. Esto será posible mediante el registro de configuración **FCDIV** que se explicará más adelante.
- Nunca se podrán borrar bloques de memoria menores que una página de 512 bytes (a excepción de las páginas 8 y 12 que son más pequeñas de lo habitual)
- Nunca se deberá programar una misma posición de memoria más de una vez después de una operación exitosa de borrado de la página que la contiene. Si se desea reescribir de nuevo esa posición, será imprescindible borrar la página entera

previamente (o borrar la memoria completa). No respetar esta restricción puede acarrear la corrupción de los datos almacenados en la Flash.

A continuación se procederá a describir resumidamente las propiedades incluidas y las herramientas disponibles en la plataforma MC13213 que tienen alguna relación con la gestión de la memoria del dispositivo y pueden influir de algún modo en el diseño final del bootloader.

4.1.2.2.1. Memoria Flash del MC131213. Algoritmos de Sobrescritura

Como se acaba de introducir, la Flash (ROM) de la plataforma MC13213 consiste en algo menos de 60 KB repartidos entre dos bloques de memoria separados por los registros de paginación alta.

Para poder emplear dicha memoria para el almacenamiento de aplicaciones, el MC13212 provee de un interfaz (integrado por registros e indicadores) que permite el borrado y sobrescritura de los datos grabados en ella. Este interfaz se utiliza durante el proceso de programación inicial del dispositivo justo antes de salir al mercado pero puede emplearse adicionalmente para actualizar más tarde los dispositivos en cualquier momento de su vida útil apoyándose para ello en un bootloader como el desarrollado a lo largo de este proyecto.

Así pues, la memoria Flash del MC13213 ofrece una serie de registros sobre los que se apoyará la aplicación del bootloader para realizar las operaciones de borrado y regrabado de ésta y cuyo esquema completo de funcionamiento se describirá más adelante en el subapartado 4.1.4.1. A continuación se describen resumidamente:

El *FLASH Clock Divider Register* (**FCDIV**) es el registro encargado de configurar el reloj interno del módulo Flash a una frecuencia de entre 150 a 200 KHz para que éste pueda funcionar correctamente. Antes de poder emitir comandos de escritura o borrado sobre la memoria no volátil de la plataforma MC13213, es imprescindible configurar previamente este registro. Este objetivo se logra programando el registro **FCDIV** con la configuración deseada a partir de la frecuencia del bus y una serie de factores de preescalado.

De este modo, el valor que el bootloader programará en el registro **FCDIV** será muy dependiente de su posición en la secuencia de arranque ya que, a lo largo de ésta, las distintas fases del arranque irán modificando el reloj de referencia del chip (aumentando de frecuencia) hasta alcanzar la frecuencia del módem radio (32 MHz de reloj y 16MHz de frecuencia de bus).

Una limitación de la Flash será que el registro **FCDIV** sólo puede ser programado una única vez entre cada reseteo del dispositivo, de modo que dicha operación suele realizarse durante el

arranque del primer módulo con capacidades de regrabado de la Flash. De ahora en adelante, el registro **FCDIV** de programará en la fase inicial de la ejecución del bootloader (pero sólo cuando haya que actualizar el *firmware* del dispositivo y se indique mediante el *flag* de regrabado activo en la EEPROM). En el caso de no realizarse una actualización del *firmware*, la configuración de dicho registro se realizará durante la inicialización del módulo **NVM** (explicado al final de este apartado), varios pasos en la secuencia de arranque más adelante.

Adicionalmente, es necesario resaltar que la configuración del registro **FCDIV** sólo deberá realizarse después de limpiar previamente los registros de estado del módulo Flash.

El *FLASH Status Register* (**FSTAT**) es el registro encargado de informar acerca del estado de procesado del último comando enviado al controlador de la memoria Flash. Así pues, su lectura permite obtener información acerca de los siguientes parámetros relacionados con la actualización de la memoria:

- Detectar si ha tenido lugar algún error en el procesado del comando (mediante lecturas de su cuarto bit, ACCERR) al no respetarse el protocolo de ejecución de un comando.
- Comprobar si el buffer de comandos está lleno o puede continuarse un proceso de escritura por ráfagas (mediante la lectura del séptimo bit, FCBEF)
- Comprobar si ha terminado el procesado del último comando solicitado (mediante la lectura del sexto bit, FCCF)
- Detectar errores causados por el intento de sobrescribir sectores de memoria Flash securizados (mediante la lectura del quinto bit, FPVIOL)

Adicionalmente, este registro permite realizar una serie de operaciones críticas relacionadas con la grabación de la memoria Flash según se escriba en un determinado bit del registro:

- Forzar el comienzo del procesado de un comando Flash (bit FCBEF a 1)
- Reinicialización del estado del registro mediante la escritura de un 1 sobre cada bit de estado que ha indicado un suceso

Antes de realizar cualquier operación sobre la Flash, será necesario comprobar y limpiar este registro, **FSTAT,** para reiniciar el módulo en el caso de presentar un estado de error (bit FACCERR).

El *FLASH Command Register* (**FCMD**) es el registro encargado de recibir y gestionar peticiones de procesado de comandos al controlador Flash. Para hacer uso de esta funcionalidad basta con escribir sobre dicho registro uno de los 5 siguientes posibles valores (cualquier otro valor daría error):

Comando Flash	FCMD
Comprobación de Página Limpia *	0x05
Escritura Estándar (1byte)	0x20
Escritura por Ráfagas (1 byte) *	0x25
Borrado de Página (512 bytes)	0x40
Borrado Completo de Flash *	0x41

***: Comandos no empleados en el bootloader (ver más adelante)**

Una vez "enviado" el comando deseado al registro **FCMD,** será necesario activar el bit FCBEF (bit número 7) del registro **FSTAT** para que éste comience a ejecutarse (como se explicó justo en los párrafos anteriores relativos al registro **FSTAT**).

Aunque el bootloader desarrollado en este proyecto emplea la "escritura estándar" en memoria Flash que se realiza byte a byte, existe otra modalidad de programación de la memoria Flash mediante el comando de "escritura por ráfagas" que emplea un algoritmo similar al primero aunque más complejo.

Las temporizaciones aproximadas de los comandos que se emplearán para actualizar la memoria Flash del MC13213 se describen en la siguiente tabla:

Parámetro	Nº de Ciclos de Reloj del Módulo Flash (F_{CLK})	Tiempo Estimado (suponiendo F_{CLK}=200KHz)
Escritura Estándar (1 byte)	9	45 µs
Escritura a Ráfagas (1 byte)	4	20 µs[1]
Borrado de Página (512 bytes)	4000	20 ms

[1] Excluyendo los incrementos de tiempo asociados a la escritura del primer byte de la ráfaga

Como puede apreciarse, el comando de escritura por ráfagas permite teóricamente reducir el tiempo total de grabación a un poco menos de la mitad pero requiere de una serie de prerrequisitos para poder aplicarlo de forma eficaz. Estos serán los siguientes:

1. Necesidad de suministrarle ristras de datos (menos de 64 bytes) contiguos
2. La ráfaga ha de pertenecer a una fila concreta de la Flash (sectorizando la memoria en 1024 filas de 64 bytes). De lo contrario el cambio de fila implicará un cambio de velocidad durante la duración de escritura de un byte a la velocidad estándar

3. Para poder mantenerse el modo de grabación por ráfagas, cada byte nuevo de datos ha de suministrarse antes de la finalización del grabado del anterior. De lo contrario el algoritmo regresa al modo estándar de grabación (con su velocidad asociada)

Si se realizan unos cálculos rápidos, en el mejor de los casos (con la mínima frecuencia de reloj configurable, 150KHz) la sobrescritura de la memoria Flash completa mediante ráfagas tardaría algo menos de 2 segundos que empleando la grabación estándar. Lo más habitual será que debido al almacenamiento en EEPROM de registros S19 modificados de menos de 32 bytes y su disposición en posiciones de memoria no consecutivas ni contenidas enteramente dentro de un mismo sector de 64 bytes, las diferencias de emplear un método frente al otro no sean muy significativas en cuanto a tiempo se refiere.

No obstante, el cumplimiento de los prerrequisitos y control de errores necesarios para realizar la grabación por ráfagas complicaría muy considerablemente el algoritmo de actualización del *firmware* llevado a cabo por el bootloader y, con ello, sus requerimientos de memoria ROM y RAM. Es por esto que este proyecto empleará el modo estándar de grabación (descartado la escritura por ráfagas) para llevar a cabo su objetivo de reemplazar el *firmware* en uso del dispositivo.

Enviar un comando (de grabación o borrado) al controlador del módulo Flash del MC13213 puede provocar errores de lectura de cualquier código almacenado en ROM (incluyendo al código relativo al bootloader) mientras dure la ejecución de la operación solicitada. Es por ello que se hace imprescindible emplazar la rutina de escritura de bytes y borrado de páginas Flash en la memoria RAM del dispositivo de forma previa al envío de cualquiera de ambos comandos al controlador. Esta característica del módulo Flash complicará el código del bootloader y aumentará los requisitos mínimos de memoria RAM y ROM necesarios para su funcionamiento.

De este modo, el bootloader desarrollado en este proyecto hará uso de las herramientas suministradas por el MC13213 para el borrado y regrabado de su memoria Flash (descartando el modo por ráfagas, como ya se ha justificado), y respetará todas las restricciones que se han ido comentando en este apartado y sus predecesores. El resultado de este desarrollo y su descripción detallada (incluyendo el algoritmo completo de borrado y actualización de la memoria Flash) tendrán lugar en el apartado 4.1.4.1 en la parte correspondiente al bloque de sobrescritura de la memoria ROM.

4.1.2.2.2. Protección de Bloques de Memoria Flash

El MC13213 provee de un mecanismo de protección de bloques de memoria Flash que previene borrados y/o reprogramaciones accidentales (o provocadas) sobre las posiciones configuradas.

Así pues el chip permite proteger un bloque del final de la memoria Flash (desde una posición determinada hasta la posición 0xFFFF), proteger la totalidad de la memoria o por el contrario desprotegerla completamente.

Con este fin, los bits del registro no volátil **NVPROT** (cuyo valor se graba en el dispositivo y ya no puede ser modificado sin previamente borrar la página entera) y los del registro **FPROT** (sito en RAM y cuyo contenido es una copia del de **NVOPT** que se realiza automáticamente durante el arranque del dispositivo) permitirán configurar el funcionamiento del mecanismo de protección de la memoria Flash.

De este modo, la configuración del MC13213 acerca de la protección de sectores de su memoria Flash vendrá fijada de forma permanente en el registro **NVPROT**, pero podrá ser temporalmente modificada mediante la edición del registro **FPROT** a través de comandos especiales enviados por el puerto de depuración (pin BKGND del chip) pero nunca desde la aplicación en ejecución. Una vez el dispositivo se reinicie se volverá a restaurar la configuración original almacenada en **NVPROT**. Es por todo esto que una vez que el dispositivo esté instalado en localización determinada y ya no se realicen procesos de depuración sobre él, mantendrá una configuración de protección de bloques que se mantendrá constante durante toda su vida útil.

Si por cualquier razón se trata de reprogramar o borrar una posición de memoria perteneciente a un bloque protegido, se activará un bit indicador del error (bit FPVIOL) en el registro **FSTAT** y el comando se descartará sin mayores consecuencias.

El funcionamiento del módulo de protección será el siguiente: un bit del registro **NVPROT** (bit FPOPEN) permitirá el regrabado de las posiciones de memoria no protegidas, otro bit (bit FDIS) activará o desactivará el mecanismo de protección de sectores y otros 3 bits (bits FPS) configurarán el tamaño del bloque de memoria protegido (en el caso de protegerse alguno).

La utilidad más interesante de esta propiedad de la memoria del MC13213 consiste en la protección del cargador de arranque (y el bootloader incluido dentro) si se almacena en un bloque de memoria próximo al final de la Flash, de forma que el bootloader pueda borrar el resto de la memoria y reprogramarla sin peligro de dañar su propio código. Adicionalmente, si

tuviese lugar algún problema de alimentación del dispositivo mientras está siendo actualizado, el cargador de arranque no sufriría nunca ningún daño y el equipo podría recuperarse.

Desafortunadamente a efectos de este proyecto, para que este sistema de protección de memoria fuese realmente eficaz a la hora de proteger el cargador de arranque del dispositivo, éste debería estar localizado íntegramente dentro del mismo bloque de memoria protegida en todas las aplicaciones diseñadas para el ND07 y no podría modificarse en ninguna de las actualizaciones que se fueran enviando al dispositivo. Esto implicaría el desarrollo de un cargador de arranque completo y hermético en ensamblador (no sólo la parte del bootloader encargada de actualizar el *firmware*) capaz incluso de asociar el dispositivo a una red ZigBee para recuperar su *firmware* en caso de error de actualización o similar. Esto puede no ser deseable en situaciones en que se desee incluir o descartar "dinámicamente" (entre distintas actualizaciones de *firmware*) funcionalidades del dispositivo (no incluir el control de LEDs, UART, etcétera) e implicaría tomar importantes decisiones de diseño acerca de cómo habrán de funcionar de ahora en adelante todas las aplicaciones ZigBee sobre el ND07, algo completamente fuera de la competencia de este proyecto.

A día de hoy el cargador de arranque del ND07, empleado en todas las aplicaciones desarrolladas para este proyecto, se basa en programas de ejemplo que suministra *Freescale* (muy funcionales y verificados) cuyo código se compila ocupando aleatoriamente la ROM del dispositivo de forma que precise de la cantidad mínima de memoria o sea lo más rápido posible (el compilador permite emplear uno u otro criterio a la hora de generar el código ensamblador final). La inclusión de todo este código junto con el bootloader y con el encargado de dejar la red ZigBee operativa (y lista para recuperar el dispositivo en caso de darse un error de actualización) forzaría a crear un bloque de memoria protegido y hermético de un tamaño muy elevado, dejando muy poco espacio libre para el diseño de la aplicación final y restringiendo muy considerablemente el funcionamiento final de ésta.

Como ya se ha mencionado en repetidas ocasiones, el objetivo final de este proyecto es diseñar y programar un bootloader óptimo, robusto y cerrado que sea a la vez lo suficientemente flexible como para que se pueda incluir en muy diversas aplicaciones creadas para el ND07 u otros dispositivos basados en el chip ND07.

Las ventajas principales de la protección de bloques consisten el proteger un cargador de arranque completo (no sólo la parte del bootloader) capaz de arrancar el dispositivo y asociarlo a la red dejándolo funcionando normalmente o preparado para recuperarse en caso de suceder un fallo de actualización. Programar y proteger este cargador de arranque dista mucho de los objetivos buscados, ya que nos forzarían a tomar decisiones muy restrictivas acerca de cómo habrá de ser el funcionamiento en red del dispositivo y esto se encuentra fuera de nuestra competencia.

Puesto que la protección sólo del bloque de memoria relativa al bootloader no plantea ventajas significativas (ya que si tienen lugar errores en la actualización de los bloques de memoria relativos al cargador de arranque, el dispositivo no podría recuperarse) y en cambio sí que plantea un considerable número de complicaciones y restricciones (forzaría a situar el bootloader siempre en una posición fija de memoria en todas las aplicaciones y a controlar muchas posibles situaciones de error adicionales), se ha decidido no incluir el mecanismo de protección de bloques de memoria en el diseño del bootloader.

Es por todo esto que de ahora en adelante, en todas las aplicaciones desarrolladas se configurarán los valores de los registros **NVPROT** (y por ende **FPROT**) de forma que siempre se deshabilite el mecanismo de protección de bloques.

4.1.2.2.3. Redirección de los Vectores de Interrupción

Puesto que toda configuración de la memoria Flash que proteja bloques de memoria implica automáticamente la protección de los vectores de interrupción y de reset (sitos en la última página de la Flash, como se explica al principio del apartado 4.1.2.2), el valor de éstos en principio ya no podría ser actualizado nunca entre distintas versiones de *firmware*. Es por ello que se hace necesario un mecanismo que permita al desarrollador modificar su contenido (o una alternativa al problema) sin desbloquear el código del bootloader presuntamente localizado dentro del bloque protegido. Con este objetivo se emplea la funcionalidad de redirección de vectores del MC13213.

Para activar la redirección de los vectores de interrupción (el vector de reset desafortunadamente no puede ser redirigido), basta con programar (a cero) un bit (el bit FNORED) del registro no volátil **NVPROT** y activar la protección de un bloque de memoria Flash (ver sub apartado anterior). Las únicas excepciones se darán con la protección completa de la memoria o la desprotección total de la Flash que no admitirán redirección de vectores alguna. Siempre que se active la protección de un bloque de Flash se protegerá automáticamente al menos la última página de la memoria, que incluye a los vectores de interrupción, al vector de reset y a la de sección de registros no volátiles, de modo que ya no podrán ser actualizados.

Una vez se activa la redirección de los vectores de interrupción (configuración fijada en el momento de grabación del primer *firmware* en el ND07), las posiciones de memoria del rango 0xFFC0 a 0xFFFD (vectores de interrupción) se redireccionarán a las posiciones de Flash desprotegidas más cercanas al bloque protegido. De esta forma, la nueva posición de los vectores de interrupción dependerá directamente del tamaño del bloque de memoria que se haya protegido previamente)

De este modo, empleando la redirección de vectores, una aplicación de bootloader sita en el bloque de memoria protegido podría reprogramar completamente la memoria Flash no protegida del dispositivo incluyendo nuevos valores para los vectores de interrupción sin modificar el código del bootloader ni los valores originales de la tabla de interrupciones.

Desafortunadamente, como puede deducirse de los manuales del chip y de los foros de discusión de *Freescale* [42], este mecanismo de redirección de los vectores de interrupción es demasiado poco flexible, ya que al mantener el bit de redirección en un registro no volátil (dentro del registro **NVOPT**, en la posición protegida 0xFFBF), no permite elegir de forma dinámica qué conjunto de vectores se empleará. Esto provoca que cuando se programa un chip con la redirección de vectores activada, si la aplicación almacenada en la sección de memoria protegida (presumiblemente un bootloader) pretende actualizar el resto de la memoria no protegida, no podrá emplear vectores de interrupción (ya que esas posiciones se borrarán y actualizarán), reduciendo drásticamente su funcionalidad.

Por otro lado, si se programase un chip sin redirección de vectores y se protegiese un bloque de memoria que incluya un bootloader, las imágenes de aplicaciones almacenadas en la EEPROM que sustituirán a la aplicación principal deberán mantener sus códigos de atención a las interrupciones en las misma localizaciones de memoria que las que tuvo el primer programa grabado en el ND07 o dejarán de funcionar. De este modo, los desarrolladores tendrían que emplear mecanismos complicados para que el compilador localizase las rutinas de atención siempre en las mismas posiciones, interfiriendo muy negativamente en su libertad y dificultando gravemente la elaboración de aplicaciones. Adicionalmente, si se protege algún bloque de memoria queda automáticamente protegido el registro que permite configurar la redirección de vectores, de forma que si una aplicación no emplea un tipo de interrupción para nada, cualquier futura aplicación que actualice a ésta primera tampoco podrá implementar posteriormente una rutina de atención para ella ya que el vector siempre estará vacío.

Así pues, el mecanismo de redirección de vectores del MC13213 no permite disponer simultáneamente de dos tablas de vectores de interrupción y alternar dinámicamente entre ellas. Para lograr un funcionamiento similar sería necesario implementar para cada vector de interrupción una pequeña rutina de atención (sito en la sección protegida de Flash) que comprobase un indicador (*flag*) que le permita decidir cuál de los vectores disponibles para esa interrupción ha de ejecutar.

Es por todo esto, y por la decisión tomada en el subapartado anterior de no proteger bloques de memoria del chip, que se ha concluido no incluir el mecanismo de redirección de vectores del MC13213 en este proyecto. El único vector de interrupción empleado por el bootloader (el asociado al bus I^2C) al emplearse únicamente para leer la EEPROM del ND07, apuntará

perennemente a la misma posición de memoria de éste (y el desarrollador habrá de respetar dicha “limitación” en todos los *firmware*s que diseñe para actualizar dispositivos ND07).

En las situaciones en que se produzcan interrupciones I^2C durante la ejecución de la aplicación principal (mientras el bootloader no se está ejecutando), éstas se deberán principalmente al proceso de recepción y almacenamiento de imágenes en la memoria auxiliar. La rutina de atención I^2C del bootloader ha sido programada para detectar estas situaciones y redireccionar excepcionalmente (de forma manual) el vector de interrupción a otra posición de memoria Flash para que el desarrollador pueda programar su propia función de control.

De nuevo pues, de ahora en adelante todas las aplicaciones desarrolladas para el ND07 habrán de configurar los registros relativos a la redirección de vectores de interrupción de forma que esta propiedad quede siempre deshabilitada.

4.1.2.2.4. Seguridad de la memoria

La plataforma MC13213 incluye circuitos para prevenir el acceso no autorizado a los contenidos de su memoria RAM y Flash.

Cuando la seguridad está activada, las memorias Flash y RAM son consideradas recursos seguros y los registros de acceso directo, registros de paginación alta y la entrada de depuración (línea por donde se depura la aplicación durante el proceso de desarrollo software) son considerados recursos inseguros.

Cuando se activa la seguridad, el acceso a la memoria RAM y Flash sólo puede realizarse de forma normal desde recursos de memoria considerados como seguros.
De este modo, un programa ejecutándose desde posiciones seguras de memoria tendrá acceso a todos los recursos disponibles del microcontrolador, así como a todas sus posiciones de memoria. Por otro lado, el intento de leer o escribir sobre dichas posiciones desde regiones (o dispositivos externos) consideradas como no seguras resultará en lecturas de ceros y escrituras ignoradas.

El microcontrolador provee de dos registros de control idénticos para activar o desactivar la seguridad del chip llamados **NOVPT** y **FOPT** sitos en la región de registros no volátil (posición 0xFFBF) y en la región de registros de paginación alta (0x1821) respectivamente.
La única diferencia entre ambos es que el primero (**NVOPT**), mantiene su valor grabado en Flash de fábrica y éste no puede ser alterado de ningún modo salvo mediante el borrado completo de la memoria no volátil. En cambio el registro **FOPT**, que almacena el valor del registro **NVOPT** que se copia desde éste durante el proceso de arranque del dispositivo,

permite modificar temporalmente el estado de la seguridad del sistema durante la duración de la ejecución de la aplicación principal (pero al resetearse el dispositivo volverá al estado original que almacenaba el registro no volátil).

Así pues, si se desea activar (o desactivar) a seguridad del dispositivo bastará con configurar de fábrica de la forma adecuada dos bits del registro **NVOPT** (bits 0 y 1, SEC00 y SEC01) y este estado se reflejará en el registro **FOPT** tras el arranque de la aplicación.

Adicionalmente, el chip provee de la posibilidad de desactivar la seguridad del chip temporalmente (hasta el próximo reseteo del dispositivo) si se activa de fábrica el séptimo bit (KEYEN) del registro **NVOPT** y el quinto bit (KEYACC**)** del registro **FCNFG**. En este caso, la aplicación deberá escribir sobre los registros **NVBACKKEY** (de la sección registros no volátiles) una clave de 8 bytes que, si coincide con la almacenada, provocará la des-securización de la memoria hasta el próximo reseteo.

El mecanismo de seguridad del ND07 permanecerá desactivado en la mayor parte de las aplicaciones por no ser realmente necesario y requerir de un estricto control de las claves almacenadas en cada aplicación. No obstante, puesto que el bootloader se ejecutará desde recursos considerados seguros como son la memoria Flash y RAM del dispositivo, su funcionamiento e implementación será independiente y no se verá alterado por la decisión del desarrollador de aplicaciones acerca de la securización (o no) del equipo.

4.1.2.2.5. El Componente NVM (*Non-Volatile Memory*)

Adicionalmente, la pila ZigBee de *Freescale* provee de un API de uso opcional para el manejo de su componente NVM (*Non-Volatile Memory*). Este interfaz permite el empleo de dos o más páginas de memoria Flash para el almacenamiento de datos (RAM) cuyo valor debe de mantenerse entre reseteos del dispositivo (contexto del equipo ZigBee en la red). Para lograr este objetivo sin agotar rápidamente el número máximo de grabaciones que permite la memoria Flash del MC13213, la API utiliza unos algoritmos especiales de ubicación de los datos, impide actualizaciones que no aporten cambios en las variables y limita la frecuencia de refresco de dichas variables (*Freescale* recomienda que las aplicaciones desarrolladas no actualicen las variables de la NVM con una frecuencia superior a una vez cada 1,8 horas para asegurar una vida del dispositivo cercana a los veinte años).

Debido a las especificaciones de la memoria Flash del MC13213, un byte de una página no puede ser sobrescrito sin antes borrar toda la página. Es por ello que para poder disponer de dos páginas destinadas al almacenamiento de datos no volátiles (configuración por defecto), será necesario reservar tres. Siempre deberá haber disponible una página extra para copiar

sobre ella la última página cuyos datos se han modificado (quedando la página que almacenaba esos datos como la nueva página libre).

Las variables de contexto almacenadas en la memoria NVM se pueden separar en los siguientes grupos (comunes para todos los dispositivos ZigBee salvo el último punto, que es particular de cada aplicación):

- Parámetros de la red a la que el dispositivo está asociado: canal, dirección del "padre" lógico, PanId, Número de hijos asociados (routers y dispositivos finales), etc.
- Tablas de dispositivos vecinos y tablas de encaminamiento de paquetes (para los dispositivos routers)
- Parámetros de seguridad y capacidades del dispositivo que se reportarán bajo petición.
- Tablas de direcciones, tablas de *bindings*, tablas de grupos y tablas de claves
- Alarmas y escenas de la *ZigBee Cluster Library*
- Datos de contexto específicos de la aplicación ZigBee particular del dispositivo (estado de salidas, de LEDs, de batería, etc.)

La ubicación del componente NVM en memoria (si la aplicación del *firmware* en uso decide utilizarlo) es relativamente configurable pero suele ocupar por defecto las **páginas 13, 14 y 15** de la Flash del dispositivo (aunque se pueden configurar más páginas de memoria con este comportamiento, no suele ser necesario).

Toda aplicación que desee ser conforme con la especificación ZigBee (cualquiera de las versiones), ha de almacenar dichos parámetros de contexto entre reseteos del dispositivo. Así pues, lo más habitual es que todos los dispositivos que carezcan de otra memoria auxiliar para almacenar dichos datos (como es el caso del dispositivo ND07, que dispone de una memoria auxiliar pero que se emplea para el almacenamiento de imágenes) hagan uso del componente NVM.

No obstante, haga uso la aplicación del componente NVM o no, las páginas empleadas para el almacenamiento de datos no volátiles son, a fin de cuentas, memoria Flash normal del MC13213 y como tal, el procedimiento a emplear por el bootloader para su borrado y sobrescritura sería idéntico al del resto de memoria ROM y no plantearía ningún problema técnico adicional. Sin embargo, la función principal de la NVM es la de mantener el contexto del dispositivo entre reseteos de éste y, como es lógico, también entre actualizaciones de *firmware*.

Es muy poco probable que el bootloader se emplee para sustituir el *firmware* de un dispositivo por otro cuya funcionalidad sea muy distinta (por ejemplo reemplazar el *firmware* de un sensor de temperatura por el de un detector de presencia) ya que el hardware del dispositivo no suele

modificarse. Lo habitual será actualizar el *firmware* sólo para añadir funcionalidades o corregir *bugs*.

En el hipotético caso de que la funcionalidad del dispositivo cambiase radicalmente en la nueva imagen, bastaría con tenerlo en cuenta a la hora de programar la nueva versión de *firmware* para que sea compatible con los parámetros de contexto almacenados en la NVM (y respetase las páginas ocupadas por ésta no ocupándolas con el nuevo *firmware*) o se encargue de reemplazar los que considere oportunos en su primera ejecución.

A efectos del bootloader sólo existen pues dos posibilidades: incluir la NVM o ignorarla. En el primer caso el bootloader protegería las páginas asociadas a la NVM y no las borraría ni sobrescribiría para que el dispositivo, una vez reseteado, recuperase su contexto tal como estaba antes de la actualización. En el segundo caso, el bootloader funcionaría como si la NVM no existiese borrando sus páginas asociadas (siempre) y sobrescribiéndolas sólo si el nuevo *firmware* aloja datos o código en ellas.

La única pregunta que el programador tendrá que hacerse será si incluir el módulo NVM dentro de una aplicación ZigBee determinada (y sus futuras revisiones) o no hacerlo en absoluto. Pero una vez tomada una decisión, ésta habrá de mantenerse en todas las actualizaciones de *firmware* del dispositivo a lo largo de su vida útil.

Así pues, los pros y contras de incluir el módulo de NVM en una aplicación ZigBee se pueden resumir en los siguientes puntos:

- La NVM es un interfaz sencillo para almacenamiento del contexto de un dispositivo sin necesidad de memorias auxiliares adicionales ni de la programación de nuevas APIs para ese fin. Además esta API está perfectamente integrada con la librería de la capa de red ZigBee, facilitando mucho el trabajo al desarrollador.
- Las imágenes almacenadas en la EEPROM serán más pequeñas, ya que las páginas correspondientes a la NVM se descartarán puesto que no se sustituirán en Flash.
- Menor espacio consumido en la EEPROM en concepto de imágenes, lo cual implica mayor velocidad del proceso de almacenamiento de *firmwares* y de actualización de la Flash (reduciendo las posibilidades de errores críticos).
- Incluir el módulo NVM en una aplicación implica emplear para ese propósito unos 2 KB de código en Flash y 150 bytes de RAM adicionales a las 3 páginas (en el mejor caso, 3x512 bytes) que habrán de reservarse para el almacenamiento de los datos no volátiles de contexto. Esto implica que como mínimo la aplicación dispondrá de casi 4KB menos de espacio en Flash para código de aplicación, y menos RAM, que son recursos muy escasos en aplicaciones complejas como routers y coordinadores. No

obstante, la inclusión de cualquier API que controle el almacenamiento de datos no volátiles en una memoria auxiliar implicará ocupar esta cantidad de memoria o más.

Puesto que las aplicaciones desarrolladas sobre el dispositivo ND07 pretenden ceñirse a la especificación ZigBee lo máximo posible (para lo cual han de almacenar los datos de contexto de la aplicación) y este dispositivo no dispone de más memoria auxiliar que la EEPROM que ya se emplea para el almacenamiento de imágenes, el empleo del componente NVM es casi obligatorio.

Desarrollar un bootloader en ensamblador que permita elegir si se desea incluir el componente NVM o no (y de incluirlo, qué páginas ocuparán los datos no volátiles de contexto) es lo suficientemente complicado (aumentando considerablemente el tamaño, lentitud y complejidad de integración del bootloader con una aplicación ZigBee) como para que no compense cubrir una situación tan poco probable como es la de descartar el componente NVM.

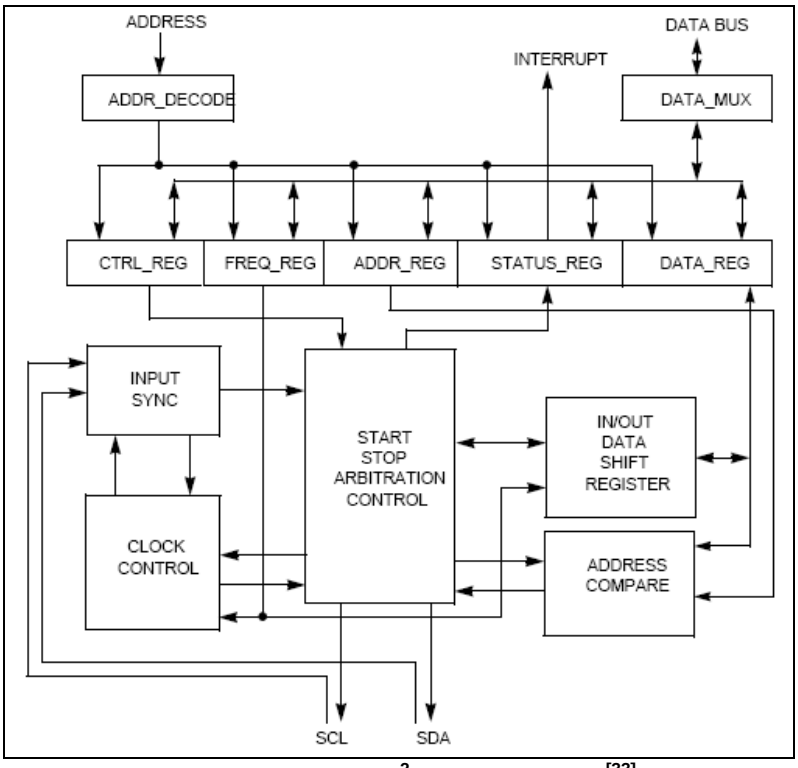
De este modo, el bootloader desarrollado respetará también las páginas correspondientes a los datos de contexto de la NVM, no borrándolas ni sobrescribiéndolas nunca.

4.1.2.3. Módulo I^2C del MC13213

El módulo I^2C incluido en la plataforma MC13213 proporciona las siguientes funcionalidades:

- Compatibilidad con el bus I^2C
- Funcionamiento “Multi-Maestro”
- 64 diferentes frecuencias de reloj serie programables
- Bit ACK seleccionable por software
- Transferencia de datos mediante interrupciones byte a byte
- Interrupción de pérdida de arbitraje con cambio automático de modo maestro a esclavo
- Detección y generación de señales START y STOP
- Generación de señal de START repetido
- Generación y detección del bit de ACK
- Detección de bus ocupado

Estas funcionalidades se ilustran en el siguiente esquema:



Bloque Funcional I²C del MC13213 [33]

El control de estas funcionalidades del módulo I²C se realiza mediante la escritura y lectura de 5 registros de 8 bits: **IICA**, **IICF**, **IICC**, **IICS** y **IICD**.

El registro **IICA** (*I²C Address Register*) almacena la dirección de 7 bits del módulo I²C que lo identifica cuando funciona en modo esclavo. Esta situación no se dará nunca en el ND07 ya que sólo existen dos componentes conectados a su bus I²C, el MC13213 y la EEPROM, que siempre funcionarán como dispositivo maestro y esclavo respectivamente.

El registro **IICF** (*I²C Frequency Divider Register*) permite configurar la velocidad del bus I²C y el tiempo que permanecen activos los bits en la línea de datos (SDA) después de que la línea de reloj haya cambiado de nivel alto a nivel bajo.

El registro **IICC** (*I²C Control Register*) otorga control sobre el funcionamiento del MC13213 en el bus. Así pues permite:

- Activar o desactivar el módulo I²C
- Alternar entre los roles de maestro y esclavo
- Alternar entre los modos de transmisión y recepción
- Decidir si se responde a una recepción mediante ACK o no se responde (NACK)
- Habilitar o deshabilitar las interrupciones I2C
- Forzar un START repetido

El registro **IICS** (*I^2C Status Register*) permite a la aplicación detectar el estado del bus o sucesos puntuales que han tenido lugar como pueden ser:

- Final de una transferencia (tanto de recepción como de transmisión)
- Un dispositivo ha iniciado una comunicación como maestro dirigida al MC13213 como esclavo (esta situación no se dará nunca en el ND07)
- Bus ocupado (si se ha escuchado un START en el canal y aún no se ha recibido el STOP)
- Pérdida de arbitraje
- Modo de funcionamiento (recepción o transmisión) del MC13213 cuando está funcionando con el rol de esclavo (esto tampoco tiene nunca lugar en el ND07)
- Interrupciones I^2C ante pérdidas de arbitraje, transferencias finalizadas de un byte de datos, o coincidencia de direcciones de dispositivos durante el proceso de direccionamiento
- Recepción (o no) de bit de ACK tras el envío de un byte de datos

La lectura del registro **IICD** (*I^2C Data IO Register*) permitirá recuperar el último byte recibido del bus I^2C (en modo recepción) y su escritura permitirá el envío de datos. La lectura o escritura sobre este registro, con los demás registros convenientemente configurados, permitirán inicializar una comunicación en modo recepción o transmisión respectivamente.

Adicionalmente, la interrupción I^2C permite a la aplicación detectar y actuar en consecuencia ante situaciones de:

- Pérdida de arbitraje del bus.
- Finalización de transferencia de un byte.
- Recepción de una trama que direcciona al MC13213 como esclavo de una transmisión (esto no sucederá nunca en el ND07).

El empleo sabio de este interfaz formado por cinco registros y una interrupción, junto los conocimientos adquiridos acerca del funcionamiento del protocolo I^2C y su implementación para la memoria AT24C512 (ver apartado 3.9) permitirán desarrollar un API para la lectura y escritura de la EEPROM externa que el bootloader empleará durante su ejecución (ver bloque de lectura de registros S19 modificados del apartado 4.1.4.1).

4.1.2.4. Vectores de Interrupción

Los diferentes módulos de la plataforma MC13213 (interfaces de comunicación serie, conversor analógico-digital, módulo SPI e I^2C, temporizadores hardware, KBI, etc.), emplean habitualmente interrupciones para indicar diferentes estados de funcionamiento.

Como es habitual, a cada interrupción le corresponde un vector de interrupción que apunta a la sección de código en memoria Flash encargada de efectuar su rutina de atención.
En el caso del microcontrolador MC13213, todos los vectores de interrupción se encuentran alojados por defecto en la última página de memoria Flash, aunque pueden redireccionarse a otras posiciones si se activa la protección de bloques de memoria (ver apartado 4.1.2.2).

El listado de las distintas interrupciones posibles y su emplazamiento por defecto en memoria Flash se describe en la siguiente figura:

Reset and Interrupt Vectors

Address (High/Low)	Vector	Vector Name
0xFFC0:FFC1 ↕ 0xFFCA:FFCB	Unused Vector Space (available for user program)	
0xFFCC:FFCD	RTI	Vrti
0xFFCE:FFCF	IIC	Viic1
0xFFD0:FFD1	ATD Conversion	Vatd1
0xFFD2:FFD3	Keyboard	Vkeyboard1
0xFFD4:FFD5	SCI2 Transmit	Vsci2tx
0xFFD6:FFD7	SCI2 Receive	Vsci2rx
0xFFD8:FFD9	SCI2 Error	Vsci2err
0xFFDA:FFDB	SCI1 Transmit	Vsci1tx
0xFFDC:FFDD	SCI1 Receive	Vsci1rx
0xFFDE:FFDF	SCI1 Error	Vsci1err
0xFFE0:FFE1	SPI	Vspi1
0xFFE2:FFE3	TPM2 Overflow	Vtpm2ovf
0xFFE4:FFE5	TPM2 Channel 4	Vtpm2ch4
0xFFE6:FFE7	TPM2 Channel 3	Vtpm2ch3
0xFFE8:FFE9	TPM2 Channel 2	Vtpm2ch2
0xFFEA:FFEB	TPM2 Channel 1	Vtpm2ch1
0xFFEC:FFED	TPM2 Channel 0	Vtpm2ch0
0xFFEE:FFEF	TPM1 Overflow	Vtpm1ovf
0xFFF0:FFF1	TPM1 Channel 2	Vtpm1ch2
0xFFF2:FFF3	TPM1 Channel 1	Vtpm1ch1
0xFFF4:FFF5	TPM1 Channel 0	Vtpm1ch0
0xFFF6:FFF7	ICG	Vicg
0xFFF8:FFF9	Low Voltage Detect	Vlvd
0xFFFA:FFFB	IRQ	Virq
0xFFFC:FFFD	SWI	Vswi
0xFFFE:FFFF	Reset	Vreset

Vectores de interrupción y reset del MC13213 [33]

A efectos del bootloader, la única interrupción imprescindible que para su correcto funcionamiento será aquella generada por el módulo I^2C (cuyo vector se encuentra en las posiciones de memoria **0xFFCE** y **0xFFCF**), fruto de la comunicación del microcontrolador con la memoria E2PROM externa (que almacena la imagen del *firmware* que sustituirá al actual).

En la plataforma MC13213, la mayoría de las interrupciones se encuentran desactivadas por defecto y no tendrán lugar a menos que el código de aplicación, o alguna instrucción de la secuencia de arranque, las habilite específicamente.
Es por esto que, para evitar al máximo los riesgos derivados de que cualquier interrupción ajena al bootloader interrumpa su ejecución repetidamente (o durante un tiempo demasiado prolongado), se recomienda ejecutar el bootloader lo más próximo posible al principio de la secuencia de arranque. De este modo, poco después del encendido del dispositivo se comprobará si existe una actualización programada en la EEPROM y, de existir, se procederá a sustituir el *firmware* mucho antes de dar paso al resto de módulos del microcontrolador o a la aplicación principal.

La localización de los vectores de interrupción en la última página de la memoria Flash (página número 127), obligará a tratar este bloque de memoria de forma especial a la hora de actualizar el *firmware* de un dispositivo. El borrado accidental de este sector sin realizar una inmediata sobrescritura de los vectores de interrupción puede provocar que una actualización se quede a medias dejando el dispositivo en un estado irrecuperable. Este punto se desarrollará con mayor profundidad en los próximos apartados.

4.1.2.5. Watchdog

La plataforma MC13213 permite la activación opcional de un temporizador hardware configurable llamado ***watchdog*** (o ***COP***) cuya función consiste en forzar un reseteo del dispositivo cuando la aplicación deja de funcionar como se espera de ella, quedándose "colgada".

Para evitar este reseteo, la aplicación ha de encargarse de reinicializar periódicamente el *watchdog* antes de que éste expire. Si por algún error de diseño o suceso inesperado el código de la aplicación falla y se pierde (*memory leaks*, punteros erróneos, situaciones no contempladas, etc.), el *watchdog* no se reinicializará a tiempo y éste reseteará el dispositivo para devolver al sistema a un punto inicial conocido.

A día de hoy la mayor parte de las aplicaciones de dispositivos que pretendan permanecer funcionando periodos prolongados de tiempo, activan el temporizador *watchdog* en su código

para protegerse ante situaciones no contempladas (difícilmente detectables a corto plazo mientras el programador desarrolla el código).

Una aplicación bien diseñada incluirá pues la activación del *watchdog* y el código necesario para reiniciarlo periódicamente sólo si la ejecución se está efectuando de acuerdo a lo esperado. De este modo un dispositivo instalado en un emplazamiento de difícil acceso no requerirá de la asistencia de un instalador para reinicializar el dispositivo si este se "cuelga".

Como ya se ha mencionado en el capítulo tres, la tecnología ZigBee está pensada específicamente para cubrir sistemas compuestos por dispositivos que permanecen encendidos periodos de hasta años y que en muchas ocasiones se instalan en localizaciones de difícil acceso. Es por ello que cualquier desarrollador de aplicaciones ZigBee inteligente incluirá, sin duda alguna, la funcionalidad del *watchdog* en todos los códigos que programe.

Al ser este el caso del ND07, el diseño del bootloader se realizará siempre contando con que el *watchdog* está activado en todas las aplicaciones ZigBee. Puesto que habitualmente toda actualización del *firmware* llevará más tiempo que el temporizador *watchdog* en expirar, será necesario que el bootloader lo reinicialice periódicamente a lo largo de su ejecución y no arriesgarse así a un reseteo inesperado del dispositivo en medio de algún proceso crítico.

En el caso de la plataforma MC13213, el *watchdog* puede ser configurado para que expire (si no se reinicializa antes) transcurridos periodos de 2^{18} o 2^{13} ciclos del bus (periodo que puede resultar muy cortos en el tiempo dependiendo de la frecuencia de reloj empleada por el chip, que será distinta dependiendo de la fase de arranque del dispositivo).

Puesto que el bootloader incluye procesos críticos cuya duración puede llegar a ser bastante prolongada, su inclusión impondrá al desarrollador el requisito de configurar siempre el *watchdog* con la temporización más larga posible (2^{18} ciclos de bus).

El manejo y configuración del *watchdog* de la plataforma MC13213 es muy sencillo y se realiza íntegramente mediante el registro **SOPT** (*System Options Register*) y el registro **SRS** (*System Reset Status Register*).

De este modo, el temporizador *watchdog* se activará escribiendo un bit a 1 (el bit *COPE*, "*COP Enable"*) del registro **SOPT** y se configurará su periodo a 2^{18} ciclos de bus escribiendo a 1 otro bit del mismo registro (el bit COPT, "*COP Timeout*").

La reinicialización del temporizador *watchdog* se realizará escribiendo cualquier valor sobre el registro de sólo lectura **SRS** (no afectando pues, al valor en él almacenado) en cualquier momento. Obviamente se recomienda que este reseteo del temporizador no se realice nunca

dentro códigos de atención de interrupciones ya que éstas podrían seguir ejecutándose periódicamente incluso cuando el programa principal falle.

El único requisito que impone este temporizador es que el registro **SOPT** ha de configurarse siempre, incluso si se desea deshabilitar el *watchdog*, al principio del arranque del dispositivo y que este proceso sólo puede realizarse una vez por cada reseteo del dispositivo (ver pasos del cargador de arranque del apartado 4.1.2).

Como resumen se concluye pues que el bootloader desarrollado en este proyecto contará con que el *watchdog* estará activado siempre (y configurado con su periodo más largo), y lo reiniciará periódicamente a lo largo de su ejecución para evitar cualquier reseteo accidental del dispositivo.

4.1.3. Formatos de *Firmware*: Registros S19 y S19 Modificados

En este apartado se desarrollará el estudio realizado acerca del formato que sigue la imagen de una aplicación ZigBee compilada para el chip MC13213 de *Freescale* y acerca del formato óptimo que se empleará para su almacenamiento en la memoria auxiliar EEPROM optimizando el espacio libre y eliminando datos redundantes o innecesarios.

4.1.3.1. Fichero *Motorola* S19

S19 es la extensión utilizada por *Motorola* para los ficheros que almacenan las imágenes binarias de sus aplicaciones [43] y se basa en el formato de fichero HEX [44] de Intel. Los ficheros Motorola S19 (*S-Record*) encapsulan la imagen a grabar como una lista de registros (líneas) donde los datos se encuentran codificados de forma que el valor de cada byte en hexadecimal se representa mediante dos caracteres ASCII (de este modo todos los datos son caracteres imprimibles). Eso quiere decir que cada línea del fichero tiene el siguiente formato:

```
Stnnaaaaddddddddddddddddddddddddddddddd … cc
```

En donde:

S	= 'S' indica que es un "S-Record" (registro) de Motorola
t	= Tipo de registro, '0'=Cabecera, '1'=datos, '9'=Fin de fichero.
Nn	= Número de bytes del registro (línea) contando los datos, la dirección destino y el *checksum* (cada byte son dos "letras" en ASCII). Así pues será el número de bytes de datos más tres.
Aaaa	= Dirección de memoria destino de una línea de datos (número hexadecimal en ASCII).
Dd	= Datos del registro (números hexadecimales en ASCII).
Cc	= *Checksum* del numero de bytes, dirección y de los datos (número hexadecimal en ASCII). Para ello se suman todos los bytes en ASCII, se realiza una operación AND con 0x00FF (truncamos a ocho bits) y se realiza el complemento a uno (operación NOT).

El campo t puede tomar otros valores que corresponderían a memorias con un espacio de direcciones más grande e implicarían un mayor número de bytes para direccionar los datos que contienen. El CodeWarrior compila imágenes de aplicaciones para el chip MC13213 en las que emplea únicamente 16 bits para las direcciones, pudiendo así direccionar hasta un máximo de 64KB de memoria.

Los ficheros .S19 generados mediante el IDE del CodeWarrior suelen contener únicamente registros de tipo S0 (cabecera), S1 (datos) o S9 (fin de fichero).
Los registros S0 sólo almacenan comentarios como podrían ser el nombre de la aplicación contenida o su versión, pero no contendrán datos del mapa de bytes de la imagen. Así pues, estos registros pueden ser empleados por la aplicación que comunique la imagen al dispositivo a actualizar para indicarle qué *firmware* es el que se trata de transmitir. Sin embargo, estos registros no se almacenarán en ningún caso en la memoria auxiliar para ahorrar todo el espacio que sea posible.

Los registros S9 indican el fin de fichero de la imagen y no aportan información ni datos adicionales, de modo que se pueden descartar incluso antes de ser transmitidos a la aplicación que enviará la imagen al receptor.

Así pues, a efectos de la imagen a ubicar en la memoria auxiliar externa EEPROM, sólo interesará la información contenida en los registros S1.

CodeWarrior genera por defecto imágenes de aplicaciones en formato S19 cuyos registros de datos (S1) no tienen nunca una longitud superior a 32 bytes. Si a esto le sumamos el hecho de que el compilador trata de agrupar el código generado de forma que la memoria se vaya rellenando de forma secuencial sin dejar posiciones libres (siempre en la medida de lo posible y si no se le fuerza un comportamiento distinto mediante directivas especiales u opciones de compilación), nos encontramos con que el S19 es un formato considerablemente optimizado y versátil para transmitir imágenes por trozos.

No obstante, puesto que los datos están codificados en hexadecimal y ASCII, el fichero S19 ocupará como mínimo más del doble del tamaño real de la imagen de la aplicación en memoria Flash, de modo que será imprescindible elegir otro formato para su almacenamiento en la EEPROM (que se evaluará en el siguiente apartado).

4.1.3.2. Formato de Almacenamiento de Imágenes en EEPROM

Puesto que la imagen de una aplicación ZigBee generada para el chip MC13213 por el *CodeWarrior for Microcontrollers v5.1* seguirá el formato de ficheros .S19 (explicado en el apartado anterior), lo más lógico será pensar que el formato de almacenamiento de la imagen en la EEPROM será similar a éste.

Así pues, aplicar una codificación parecida a la empleada en los ficheros S19 para el almacenamiento de imágenes en la EEPROM revertiría en una considerable reducción de las necesidades de capacidad de procesamiento por parte de la aplicación ZigBee, lo cual simplificaría el código final reduciendo su tamaño (ambas características muy deseables).

Un fichero S19, al estar compuesto por números hexadecimales en ASCII (por ejemplo el byte 0xAA figuraría como dos caracteres: “AA”) aparte de las cabeceras (Sx) y los *Checksum* (en ASCII también) se puede descomponer completamente en caracteres imprimibles. Esa propiedad facilita enormemente su tratamiento a la hora de encapsular registros S19 para su transmisión y procesamiento.

Es por ello muy sencillo implementar un protocolo serie (RS232) que envíe como tramas registros S19 añadiéndoles una cabecera y fin de trama no imprimibles que garantice la

integridad de los datos con muy poco procesamiento adicional. Adicionalmente, el hecho de tener la imagen separada en bloques de datos cada uno de un tamaño máximo de 32 bytes (ocupando 64 bytes) facilita su envío de forma inalámbrica ya que requiere de muy poco procesamiento por parte de la aplicación. Hay que tener en cuenta que la longitud máxima del campo de datos de un paquete ZigBee, incluso incluyendo una configuración de alta seguridad, es superior a 64 bytes.

No obstante, esta misma propiedad provoca que el tamaño de los ficheros S19 sea más del doble del tamaño real que ocupa la imagen en la Flash de un dispositivo ZigBee. Puesto que muchas aplicaciones ocupan casi toda la memoria Flash disponible de un dispositivo ZigBee, el empleo del formato S19 para almacenamiento de imágenes en la EEPROM auxiliar es inviable ya que éstas habitualmente no cabrían.

Otra posible opción podría ser almacenar la imagen en la EEPROM con el formato exacto que ocupará luego en la Flash, byte a byte. Puesto que la EEPROM AT24C512 dispone de 64KB de memoria no existirían problemas de espacio. Desafortunadamente este modelo implica una serie de inconvenientes graves:

- Siempre se ocuparían 60KB de EEPROM (el tamaño máximo de puede ocupar una imagen en Flash), por muy pequeña que sea realmente la imagen del *firmware* almacenado. Esto obligaría a sobrescribir siempre casi todas las páginas de la EEPROM acortando su vida útil y ralentizando el procedimiento de actualización al máximo.
- Al tener que sobrescribirse siempre como mínimo 60KB de EEPROM, se maximizará el riesgo de errores de lectura o escritura sobre la memoria auxiliar.
- Al tener que almacenar la imagen tal como se copiará en Flash, incluyendo los bytes que no contienen información de la aplicación, la aplicación encargada de recibir el *firmware* y almacenarlo habrá de cumplir uno de los dos siguientes requisitos:
  a) Es capaz de recibir un gran volumen de datos en muchos paquetes grandes (en el caso de envíos de trozos de la imagen de 60KB por el canal) con el riesgo de pérdida, corrupción y reintento de paquetes que implica (y saturación de canal)
  b) Es capaz de recibir los datos en trozos que sólo contienen secciones “útiles” de la imagen y luego reensamblar la imagen final rellenando con bytes 0xFF los huecos no utilizados. Esto implica una menor ocupación del canal pero mucha más inteligencia en la aplicación de recepción, lo que se traduce en una sub-aplicación dedicada a recibir la imagen que ocupará más memoria Flash, reduciendo el tamaño máximo posible de la aplicación ZigBee principal (la parte que trabaja de forma transparente al bootloader y al código de adquisición de la nueva imagen).

Así pues de entre los dos extremos, almacenar la imagen en EEPROM tal y como se copiará a la Flash o guardarla como registros S19, se ha optado por implementar una solución intermedia. Ésta consistirá en archivar en la EEPROM los registros S19 de la nueva imagen alterando previamente su formato para reducir la cantidad de memoria que requieren.

El formato de estos nuevos registros S19, que de ahora en adelante se denominarán "**registros S19 modificados**", será el siguiente:

2 bytes	1 byte	Variable (1 – 32 bytes)
Dirección Flash Destino	**Longitud**	**Datos**

Formato de un registro S19 modificado

El campo **Dirección Flash Destino** será un número entre 0 y 65535 que apuntará a la posición de memoria Flash del MC13213 sobre la que se escribirán los datos del último campo del registro. En el registro S19 original, al estar este campo como número hexadecimal en ASCII, ocupaba el doble de bytes.

El campo **Longitud** será un número entero de un byte cuyo valor oscilará entre 1 y 32, dependiendo de la longitud del campo de datos del registro S19 modificado.

El campo de **Datos** incluirá todos los datos del registro S19 original pero sin formato alguno (se abandona la codificación como números hexadecimales en ASCII). Esto quiere decir que el campo de datos del registro modificado S19 tendrá la mitad de la longitud que tenía el mismo campo en el registro S19 original.

El campo de **Checksum** no se hereda en el registro S19 modificado porque se presupone que todos los registros S19 originales (o una versión modificada de éstos pero que incluye CRC) superan exitosamente la comprobación de CRC antes de modificarse y almacenarse en la EEPROM, de forma que la inclusión de este campo en cada nuevo registro sería redundante.

No obstante, sí que existen dos bytes de CRC de la imagen completa sitos en la cabecera de la EEPROM que permitirán comprobar la integridad del *firmware* en un momento dado (por ejemplo si se desea reenviar una imagen almacenada en EEPROM a otro u otros dispositivos para provocar una actualización masiva de equipos).

De este modo, para tener un mayor control de las imágenes almacenadas en la EEPROM y su integridad, se ha diseñado una cabecera de 14 bytes sita en las posiciones más bajas de memoria que contiene información adicional asociada a éstas. El formato es el siguiente:

0x0000 – 0x0001 (2 bytes)	0x0002 – 0x0009 (8 bytes)	0x000A – 0x000B (2 bytes)	0x000C – 0x000D (2 bytes)
Flag de Regrabación	**Versión *Firmware***	**Tamaño Imagen**	**CheckSum**

Formato y direccionamiento de la cabecera de la EEPROM

El ***Flag* de Regrabación** es un campo del que ya se ha hablado profusamente en apartados anteriores. Su función será la de indicar al bootloader, durante el proceso de arranque del dispositivo, que se encuentra almacenada en la EEPROM una nueva imagen de *firmware* lista para sustituir a la aplicación en ejecución. Cuando este campo almacene el valor **0xAABB** significará que el *flag* está activo y que la Flash se regrabará en el próximo reseteo. Cuando almacene cualquier otro valor, el *flag* se considerará inactivo.

La **Versión del *Firmware*** es un campo formado por 8 bytes que pretenden identificar unívocamente la imagen concreta que se encuentra almacenada en la EEPROM. De este modo, una aplicación distribuidora de imágenes podría consultar a otros dispositivos receptores acerca de qué versión de *firmware* tienen instalada y sugerirles (si procede) la conveniencia de iniciar el proceso de transmisión de una nueva imagen para sustituir a la antigua. Los dispositivos que reciban una imagen podrían a su vez convertirse en los transmisores de ésta hacia otros dispositivos más alejados logrando así una actualización masiva y eficiente de los equipos de una red. Obviamente la aplicación puede optar por no rellenar este campo y almacenar esta información en RAM, pero un reseteo del dispositivo (por falta de alimentación o cualquier otro motivo) la eliminaría de la memoria.

La especificación del formato de la versión del *firmware* y el modo en que éste se adjunta a cada imagen para su transmisión (ya sea dentro de registros modificados o mediante comandos especiales) es competencia de la aplicación encargada de difundir imágenes y queda fuera del alcance de este proyecto. A efectos del bootloader este campo es completamente transparente ya que sólo actualizará la memoria si el *flag* de regrabación está activo, independientemente de la versión del *firmware* en EEPROM.

El **Tamaño de la Imagen** indica el número total de bytes que ocupa la imagen almacenada en la EEPROM en concepto de registros S19 modificados (sin contar pues los 14 bytes de la cabecera). De este modo, este número incluye todos los bytes de los registros almacenados repartidos entre campos de datos, direcciones y longitudes.

Este campo permitirá a la aplicación que almacene el *firmware* en la EEPROM tener otro mecanismo de validación de la integridad (en este caso a través del número exacto de bytes) de la imagen guardada. Adicionalmente en el caso de que el dispositivo pretenda redistribuir la imagen de la EEPROM, este campo le permitirá calcular con antelación el número de paquetes ZigBee que habrá de enviar así como le indicará hasta qué posición de memoria puede leer

(hay que tener en cuenta que la EEPROM puede almacenar en el espacio no ocupado por la imagen bytes aleatorios o trozos de anteriores *firmwares* que deben ignorarse).

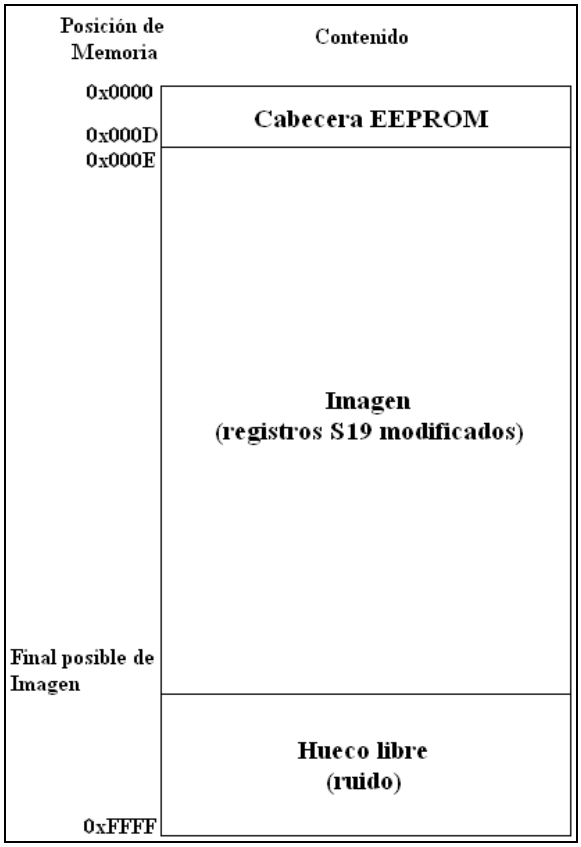
El **Checksum** permite realizar un control adicional de la integridad del *firmware* almacenado en la EEPROM. La aplicación inalámbrica que desee distribuir la imagen a otros dispositivos o quiera verificar que el almacenamiento de ésta en su EEPROM se ha realizado de forma exitosa deberá calcular el *checksum* de la imagen y compararla con el valor de este campo. La decisión acerca de cuál será el algoritmo empleado para codificar y comprobar el *checksum* la habrá de tomar el programador de la aplicación ZigBee principal. En todo caso, se recomienda el empleo de un algoritmo sencillo para minimizar así los requerimientos de memoria RAM y ROM para su implementación. Unas propuestas interesantes podrían ser la suma del valor de todos los bytes de la imagen o la aplicación de la operación matemática “O exclusivo” (XOR) a todos ellos.

Como puede deducirse fácilmente de la explicación de la cabecera de la EEPROM, todos los campos están reservados para comprobaciones que la aplicación ZigBee principal puede implementar (o no) sobre las imágenes almacenadas a excepción del *flag* de regrabación. Sólo este último afectará al bootloader que funcionará de forma transparente al valor del resto de campos. Esto se debe principalmente a las siguientes razones:

1. El bootloader, al ser el componente común de múltiples procedimientos posibles de actualización de dispositivos, ha de permanecer lo más sencillo posible. De este modo ocupará poca memoria (pudiendo incluirse dentro de aplicaciones complejas que ocupen mucho), y será más rápido y robusto (cuanto menor sea el número de lecturas y escrituras en Flash y EEPROM, menor será la probabilidad de que suceda un fallo crítico).
2. Dejar la mayor parte de las comprobaciones a la aplicación principal de carga remota permite una mayor flexibilidad en el diseño de la aplicación de actualización, permitiéndole elegir el formato del *crc* y el control de versiones más adecuado para el entorno con que se trabaje.
3. El bootloader parte de la premisa de que la aplicación de carga remota ha depositado una imagen cuya integridad ha sido convenientemente comprobada antes de activar el *flag* de regrabación. Esto se debe a que el bootloader no tiene control sobre el modo en que se adquiere la imagen (orden de paquetes, *checksums*, longitudes de tramas, etc.) y las comprobaciones que puede hacer serán siempre mucho menos exhaustivas (y muy probablemente redundantes) en comparación con las que la aplicación principal puede realizar.

Una vez se ha elegido el formado de almacenamiento de los datos del *firmware* en la EEPROM (mediante registros S19 modificados) y se ha descrito la cabecera que se incluirá al principio de

ésta, se puede esquematizar el mapa de memoria resultante de la EEPROM AT24C512 mediante la siguiente figura:



Mapa de memoria de la EEPROM

Llegados a este punto, es necesario realizar una serie de comprobaciones acerca del tamaño estimado de las imágenes de aplicaciones almacenadas en la EEPROM en forma de registros S19 modificados para evaluar si el formato elegido es suficientemente óptimo o si es necesario modificarlo o introducir alguna limitación adicional al tamaño de la imagen.

Si se examina el modelo de memoria Flash del microcontrolador incluido en la plataforma MC13213 (ver apartado 4.1.2.2), se deduce fácilmente que una aplicación ZigBee puede llegar a ocupar un máximo de 60KB de memoria (61440 bytes). No obstante, por lo menos tres páginas de ella estarán ocupadas por el bootloader (3x512 bytes) y no se almacenarán en la EEPROM ya que no se sustituirán en Flash.

Según esto, el tamaño máximo (en bytes) que la imagen de una aplicación ZigBee puede llegar a ocupar en la memoria Flash del MC13213 (excluyendo el tamaño asociado al bootloader, 3 páginas, que sólo se grabarán en el dispositivo una vez justo al final del proceso de producción) será:

$$\text{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = Flash_{TOTAL} - Tamaño_{BOOTLOADER}$$
$$\text{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = 61440 - 3 \cdot 512$$
$$\text{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = 59904\ (bytes)$$

$$N^{o}\ registros\_S19_{aprox} = \frac{\text{Im}agen\_App\_ZigBee\_en\_Flash_{MAX}\ \ (bytes)}{32\ \ \left(\frac{bytes}{registro_{S19}}\right)} = 1872\quad (registros_{S19})$$
$$Cabecera_{S19\_\text{mod}ificado} = Campo_{Dirección} + Campo_{longitud\_S19} = 2 + 1 = 3\quad (bytes)$$
$$Incremento\_\text{Im}agen\_en\_EEPROM = N^{o}\ registros\_S19_{aprox} \cdot Cabecera_{S19\_\text{mod}ificado} = 1824 \cdot 3 = 5616\quad (bytes)$$

$$\text{Im}agen\_App\_en\_EEPROM_{Aprox} = \text{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} + Incremento\_\text{Im}agen\_en\_EEPROM$$
$$\text{Im}agen\_App\_en\_EEPROM_{Aprox} = 65520\quad (bytes)$$

Este resultado ilustra el tamaño que ocuparía almacenada en EEPROM (con el formato elegido) una imagen de *firmware* que ocupase la totalidad de la memoria Flash disponible en un dispositivo. A este resultado habría que sumarle los 14 bytes de la cabecera EEPROM y se tendría un total de 65534 bytes, justo dos bytes menos que la capacidad total de la EEPROM, de modo que se dispondría de espacio suficiente para su almacenamiento.

No obstante, existen algunos casos más desafortunados que éste (aquellas imágenes con menos datos pero con muchos huecos de menos de tres caracteres entre ellos), en los cuales la imagen podría no caber en la EEPROM por muy pocos bytes. Estos casos son muy poco probables ya que el compilador optimiza la colocación de los datos en la memoria minimizando los huecos vacíos entre ellos a menos que sean muy grandes.

Sin embargo, como se explicó en el capítulo 4.1.2.2, se espera que toda aplicación ZigBee desarrollada para el dispositivo ND07 emplee parte de su memoria Flash para el almacenamiento de variables de contexto. Eso quiere decir que:

- Tres páginas se reservarán para el almacenamiento dinámico de datos de contexto no volátiles (lo cual implica que el S19 generado por el CodeWarrior no tendrá registros asociados a esas posiciones de memoria ya que se rellenarán de forma dinámica durante el tiempo de ejecución de la aplicación del dispositivo).
- Adicionalmente, por lo menos 2KB de Flash y 150 bytes de RAM del dispositivo se reservarán para el código de control del componente NVM (hay que tener en cuenta que cualquier API desarrollado para el almacenamiento de datos no volátiles ocupará esta cantidad de memoria o más). El código asociado a este API sí que estará incluido en el *firmware* distribuido mediante registros S19.

Así pues, será necesario modificar las ecuaciones anteriores:

$$\mathrm{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = Flash_{TOTAL} - Tamaño_{BOOTLOADER} - Datos\_de\_Contexto_{NVM}$$
$$\mathrm{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = 61440 - 3 \cdot 512 - 3 \cdot 512$$
$$\mathrm{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} = 58368\,(bytes)$$

El tamaño máximo de la imagen en memoria Flash se ha reducido lo suficiente como para que al realizar la aproximación de repartirlo en registros modificados de tamaño máximo (32 bytes de datos y 3 de cabecera) siga quedando espacio libre como para que aumente considerablemente el número de registros ocupados (debido a la no homogeneidad de la distribución de datos dentro del *firmware*) y siga cabiendo holgadamente.

$$N^{o}\,registros\_S19_{aprox} = \frac{\mathrm{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} \quad (bytes)}{32 \quad \left(\frac{bytes}{registro_{S19}}\right)} = 1824 \quad (registros_{S19})$$

$$Cabecera_{S19\_\mathrm{mod}ificado} = Campo_{Dirección} + Campo_{longitud\_S19} = 2 + 1 = 3 \quad (bytes)$$
$$Incremento\_\mathrm{Im}agen\_en\_EEPROM = N^{o}\,registros\_S19_{aprox} \cdot Cabecera_{S19\_\mathrm{mod}ificado} = 1824 \cdot 3 = 5472 \quad (bytes)$$

$$\mathrm{Im}agen\_App\_en\_EEPROM_{Aprox} = \mathrm{Im}agen\_App\_ZigBee\_en\_Flash_{MAX} + Incremento\_\mathrm{Im}agen\_en\_EEPROM$$
$$\mathrm{Im}agen\_App\_en\_EEPROM_{Aprox} = 63840 \quad (bytes)$$

$$Espacio\_Ocupado\_en\_EEPROM_{Aprox} = \mathrm{Im}agen\_App\_en\_EEPROM_{Aprox} + Cabecera_{EEPROM} = 63840 + 14$$

$$Espacio\_Ocupado\_en\_EEPROM_{Aprox} = 63854 \quad (bytes)$$

Así pues, se pueden concluir las siguientes afirmaciones:

- El formato elegido para almacenar la imagen en EEPROM, mediante registros S19 modificados, permite almacenar casi todas las imágenes posibles para este chip. Las posibles excepciones implicarían que el programador forzase explícitamente al compilador a distribuir los datos de la imagen en Flash de forma subóptima.
- Por otro lado, para la mayoría de imágenes posibles de aplicaciones ZigBee, este formato optimiza el espacio empleado para su almacenamiento en EEPROM, acelerando el procedimiento de actualización del dispositivo y minimizando las probabilidades de que tengan lugar fallos críticos.
- Para protegerse ante todas las posibles situaciones (por improbables que sean), **la aplicación encargada de almacenar en EEPROM la imagen recibida deberá comprobar que ésta, distribuida en registros S19 modificados, ocupa menos de 63840 bytes en total**. De lo contrario no activará el *flag* de regrabación puesto que no se podrá almacenar en su totalidad (ni se distribuirá a otros dispositivos).

4.1.4. Desarrollo del Bootloader

4.1.4.1. Estructura de la Aplicación

Como ya se ha comentado anteriormente, el bootloader desarrollado consiste principalmente en la modificación del cargador de arranque incluido en el *firmware* del dispositivo de forma que incluya una nueva etapa.

La nueva etapa programada será capaz de sustituir el *firmware* residente en la memoria no volátil (Flash) del dispositivo por una nueva imagen previamente almacenada en la EEPROM auxiliar. Este reemplazo no afectará a las páginas de memoria en las que resida esta nueva etapa del cargador de arranque aunque sí puede afectar a las demás.

El funcionamiento de este nuevo cargador de arranque se ilustra en la siguiente figura:

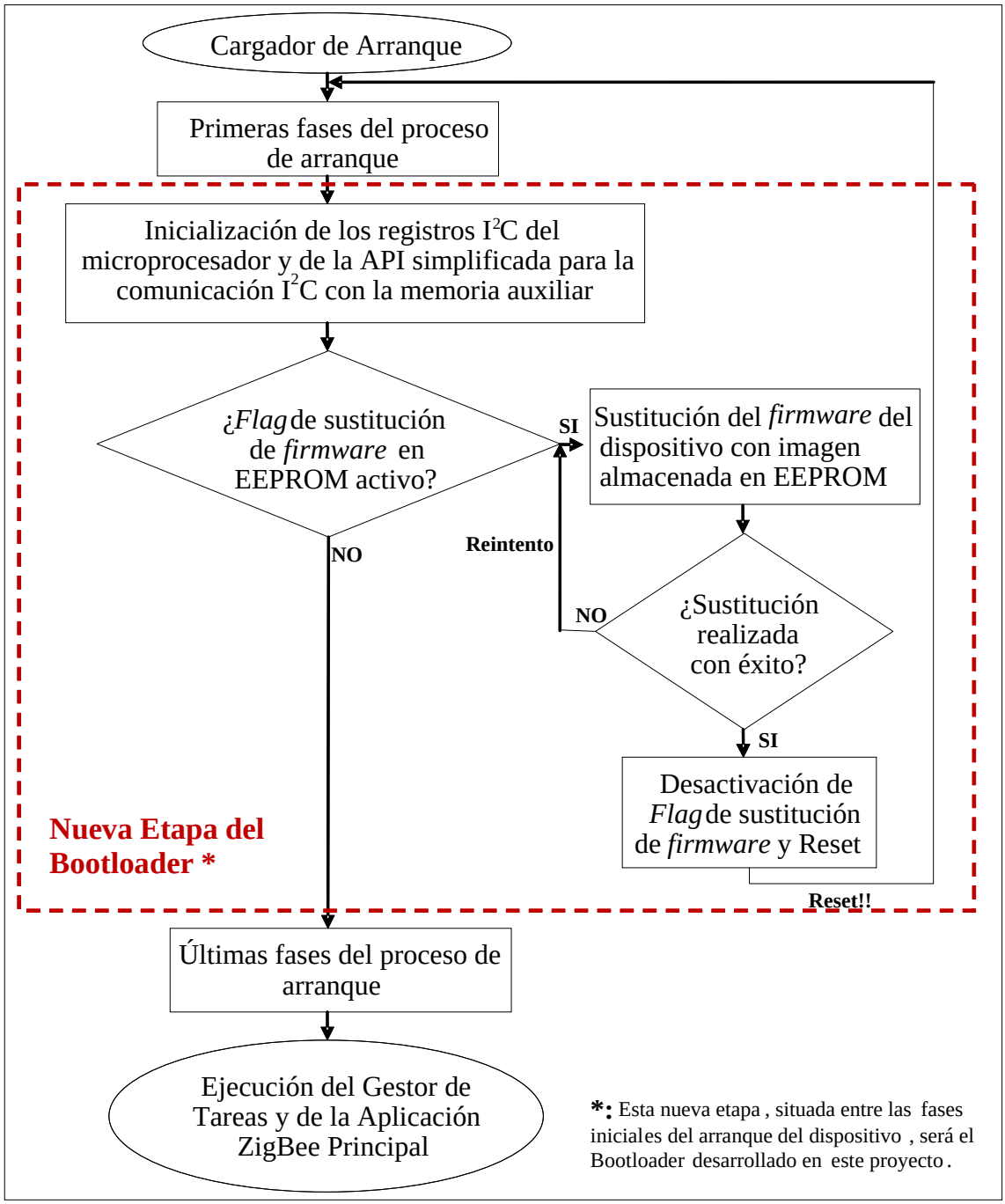


Diagrama de la estructura de la aplicación de bootloader

Como puede observarse en el esquema anterior, la nueva etapa del bootloader no modificará sustancialmente el proceso de arranque del dispositivo en las situaciones en que no exista en la memoria auxiliar un *firmware* listo para sustituir al actual (hecho que se indicará con el *flag* en EEPROM cuando sea necesario).

Así pues, en la mayoría de los casos en que el dispositivo arranque, la nueva etapa del bootloader sólo se encargará de inicializar los puertos imprescindibles para leer (vía protocolo

I^2C) el *flag* de regrabación situado EEPROM y continuará después la inicialización de forma transparente al resto de modificaciones introducidas.

Una decisión de diseño importante a tomar en este punto fue la conveniencia de situar el *flag* de regrabación en la memoria externa (EEPROM) accesible mediante el protocolo I^2C en lugar de en la memoria interna del dispositivo (Flash). Esto se debió principalmente a los siguientes factores:

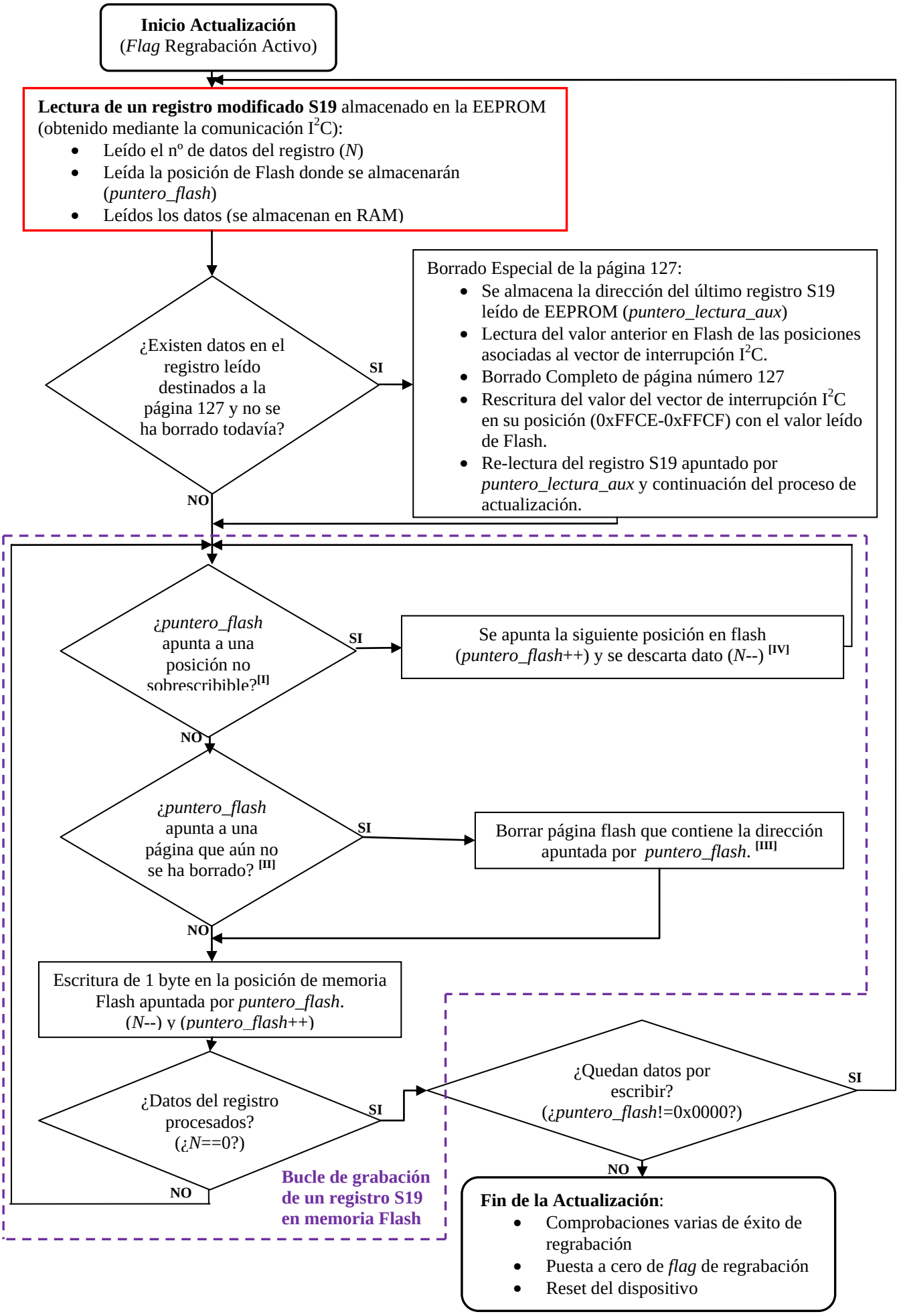
- La grabación de un byte en una posición al azar de la memoria ROM interna implica el borrado previo de la página completa que la contiene (512 bytes). De modo que se desperdiciaría demasiada memoria ROM para almacenar el *flag* de grabación (uno o dos bytes).
- La memoria ROM de *Freescale* dispone de 2 páginas por defecto utilizables para el almacenamiento de variables cuyo valor debe ser conservado entre los distintos reseteos del dispositivo (para disponer de 2 páginas con ese fin es necesario reservar 3). Estas páginas simulan el funcionamiento de una memoria RAM no volátil (aumentando el número máximo posible de regrabaciones mediante algoritmos especiales) sobre una memoria ROM (ver apartado 4.1.2.2). Desafortunadamente se han detectado y confirmado varios errores en las funciones asociadas a este componente NVM (*Non-Volatile Memory*) dentro de las librerías de *Freescale* que provocaban un funcionamiento irregular. Así pues, se ha descartado el empleo de la NVM (y de la API asociada que *Freescale* suministra para su manejo) para el almacenamiento del *flag* de regrabación.
- La elaboración de la nueva etapa del bootloader ha requerido del desarrollo de una API simplificada para la comunicación I^2C con la EEPROM externa. De este modo la programación de la comprobación del *flag* de regrabación en EEPROM es casi inmediata.

El inconveniente claro de esta elección es la necesidad de inicializar el módulo I^2C del chip y realizar una lectura de la EEPROM externa en una de las primeras fases del arranque, con el consecuente incremento del tiempo total de arranque del dispositivo. No obstante, dada la relativamente alta velocidad de la comunicación I^2C que se ha empleado, este tiempo adicional es pequeño (varios milisegundos).

Otra decisión de diseño realizada ha sido que para que la aplicación indicase al bootloader que existía un *firmware* en la EEPROM preparado para sustituir al actual, el *flag* de regrabación debía de almacenar el valor **0xAABB** (valor poco probable aleatoriamente con muchos unos y ceros en binario). Así pues se emplearon 2 bytes en lugar de uno para minimizar al máximo la probabilidad de la aparición de un falso positivo.

Una vez la aplicación principal ha decidido actualizar su *firmware* y ha almacenado la imagen de la nueva aplicación en la memoria auxiliar EEPROM (respetando el formado especificado en el apartado 4.1.3), procederá a verificar su integridad. Si esta operación se finaliza exitosamente, la aplicación activará el *flag* de actualización del *firmware* y reseteará el dispositivo para que comience a actualizarse (de este modo la aplicación y el bootloader pueden funcionar de forma casi completamente independiente).

El proceso de actualización del *firmware* de un dispositivo ZigBee es la parte más importante y crítica del proyecto que, a pesar de todas las restricciones que deben respetarse para su correcto funcionamiento, puede esquematizarse de forma simplificada mediante la siguiente figura:



Algoritmo de actualización del *firmware* del ND07

[I]: Las posiciones de memoria ocupadas por el bootloader, la NVM y el vector de interrupción de I^2C no se sobrescriben en el bucle principal. Esto se debe a que las páginas ocupadas por el bootloader y la NVM nunca se actualizarán y el vector de interrupción sólo se sobrescribirá en la fase final (proceso etiquetado como "Borrado Especial de la página 127" en el esquema anterior) cuando se lea un registro modificado S19 con datos para la última página de la memoria Flash.

[II]: Para poder escribir un dato en una posición de memoria Flash es imprescindible borrar previamente la página completa que la contiene. Puesto que un registro modificado S19 puede incluir datos pertenecientes a páginas distintas, es necesario realizar esta comprobación con cada byte a escribir para evitar fallos de actualización de la memoria.

Adicionalmente, se podrían borrar todas las páginas vacías (aquellas para las que no existan registros con datos en la EEPROM) para limpiar la memoria lo máximo posible. Sin embargo, este procedimiento adicional no se realizará ya que reduce innecesariamente la vida útil de la memoria Flash (que está limitada en cuanto al número máximo de ciclos de escritura que se pueden efectuar sobre ella) y aumenta el tiempo total necesario para finalizar el proceso de actualización.

[III]: Este apartado ha de incluir la comprobación de si la página a borrar es la número 8 o la número 12, ya que éstas requieren de un direccionamiento especial para su reinicialización (esto se debe a que no son páginas Flash completas de 512 bytes).

[IV]: En el hipotético caso en el que la aplicación encargada de recibir y almacenar el *firmware* no haya descartado las páginas relativas al bootloader (que obviamente no se sobrescribirá) los datos relativos a éstas se ignorarán.

Según la figura anterior podemos distinguir los siguientes bloques fundamentales:

- Bucle principal, encargado de leer registros S19 desde la EEPROM
- Bucle secundario, encargado de borrar cada página de memoria Flash y sobrescribirla byte a byte con los datos leídos de cada registro modificado S19
- Bloque de tratamiento especial de la página especial 127
- Bloque de comprobaciones finales y Reset

A continuación se procede a describir estos bloques con un poco más de profundidad.

4.1.4.1.1. Bucle de Lectura de Registros S19 Modificados

Este bloque funcional del Bootloader es el encargado de la recuperación del *firmware* almacenado en la memoria auxiliar EEPROM con el formato "S19 modificado" que se describe en el apartado 4.1.3.

Este bucle de lectura de la EEPROM se repetirá al menos tantas veces como registros modificados S19 almacene la memoria auxiliar y cada iteración implicará, en la mayoría de los

casos, una llamada al bucle secundario para la grabación de los datos leídos sobre una posición concreta de la memoria Flash integrada en el chip.
En este proceso, al igual que en el resto de los bloques que componen el bootloader, se realizan una serie de suposiciones que han de cumplirse para que la lectura concluya de forma exitosa (estos supuestos no se comprobarán en el bootloader para reducir al máximo el tamaño de la memoria ocupada por éste). Así pues lo prerrequisitos serán los siguientes:

- Los registros modificados S19 han de almacenar datos para direcciones de memoria Flash ordenados secuencialmente de forma creciente, de manera que el registro con datos correspondientes a la dirección de índice inferior (como mínimo 0x1080) será el primero de la EEPROM y el registro que contenga el vector de interrupción de Reset (correspondiente a las posiciones 0xFFFE-0xFFFF de la Flash) será el último.
- La aplicación encargada de almacenar el nuevo *firmware* en la EEPROM (fuera del ámbito de este proyecto) debe asegurar la integridad de los registros de datos (mediante mecanismos de *checksum*, por ejemplo), la optimización de su almacenamiento (eliminando registros asociados a páginas que no se sobrescribirán como las del bootloader) y la completitud de la imagen (que no falte ningún registro necesario).
- La imagen almacenada debe ocupar como mínimo dos registros S19 (cosa más que probable puesto que con 64 bytes es prácticamente imposible diseñar una aplicación ZigBee).
- Cada registro modificado S19 no debe almacenar más de 32 bytes de datos (y ha de contener al menos un byte de datos) y su campo de longitud debe almacenar el tamaño exacto del número de bytes de ese registro.

Todos estos requisitos son muy fácilmente alcanzables ya que la configuración por defecto del IDE *CodeWarrior for MicroControllers v5.1*, que es la herramienta suministrada por *Freescale* para compilar códigos ZigBee para este chip, genera los registros en el formato especificado (en cuanto a tamaño y número de registros, orden de éstos según la dirección apuntada, etcétera) de modo que no deberían plantear ningún problema. Por otro lado, los requisitos de integridad exigidos a la aplicación que almacenará el nuevo *firmware* en la EEPROM son los mínimos que un proceso tan crítico como es la sustitución del programa en ejecución de un dispositivo requerirá siempre.

Como ya se ha mencionado con anterioridad, la comunicación con la EEPROM auxiliar integrada en la placa del ND07 (modelo AT24C512) ha requerido de la programación de una versión simplificada del protocolo I^2C para poder realizar lectura y escrituras sobre ella.

Puesto que el ND07 sólo dispone de dos dispositivos conectados al bus I^2C, el chip MC13213 funcionando siempre como maestro y la EEPROM funcionando siempre como equipo esclavo, situaciones como la pérdida de arbitraje no se darán nunca, ya que en ningún caso varios dispositivos en modo maestro competirán por el bus. De este modo, no sería necesario

implementar el protocolo completo para lograr una perfecta comunicación entre los dos integrados.

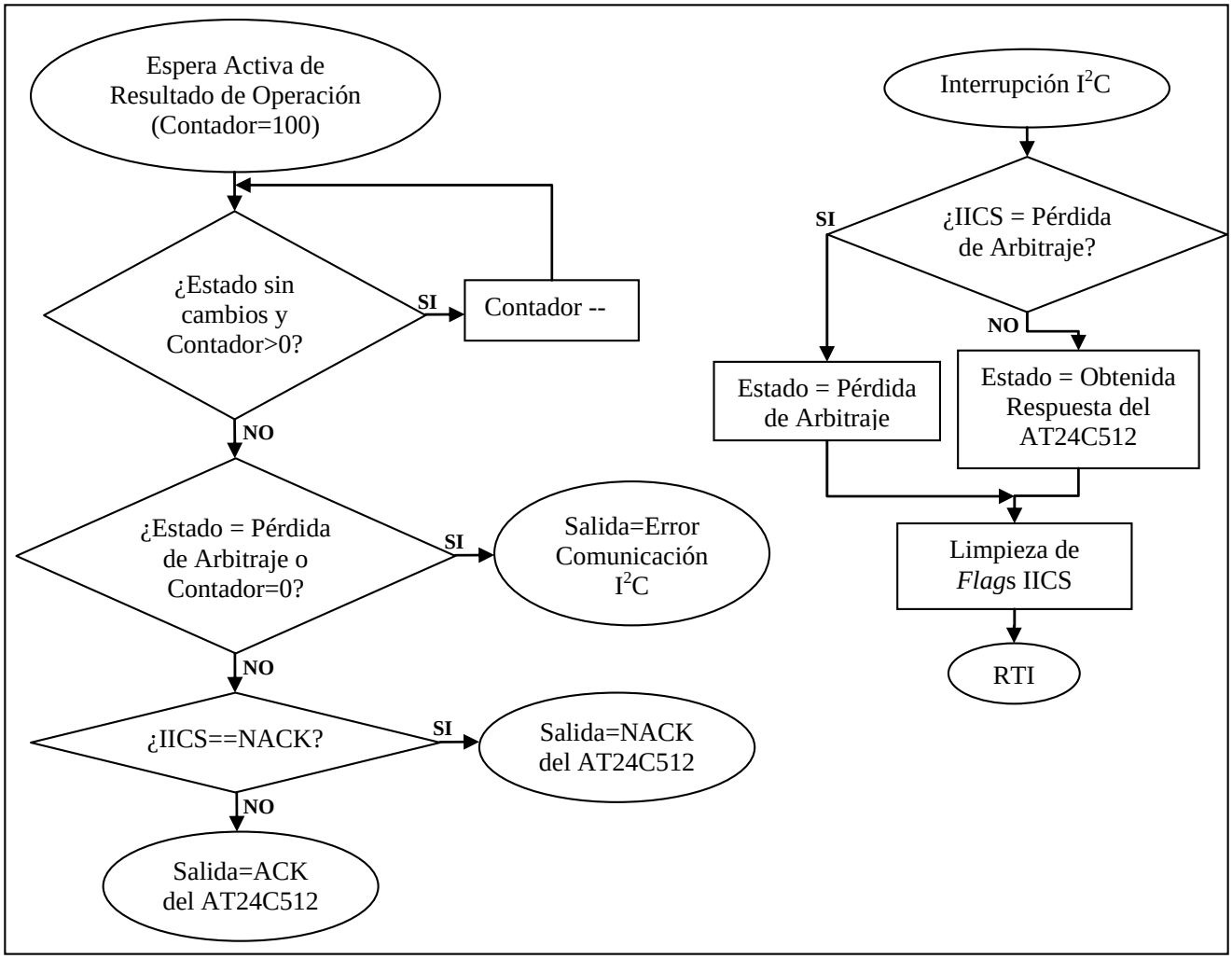
No obstante, se desarrolló un API simplificado que cubre el protocolo completo I^2C puesto que el módulo del MC13213 estaba específicamente preparado para ello y no requería de un esfuerzo significativo adicional ni incrementaba el tamaño del código resultante de forma apreciable. Para ello ha sido necesario apoyarse en el interfaz que otorga el módulo I^2C de la plataforma MC13213 mediante los registros descritos en el apartado 4.1.2.3 y sus interrupciones asociadas.

Previamente fue necesario desarrollar dos funciones auxiliares:

- La función de atención a la interrupción I^2C: configurada para activarse ante cualquiera de los siguientes sucesos en el bus:
    1. Detección de pérdida de arbitraje
    2. Finalización de transferencia de un byte (mediante la recepción de ACK o NACK por parte del dispositivo esclavo)
    3. Recepción de una trama que direcciona al MC13213 como esclavo de una transmisión (esto no sucederá nunca en el ND07 ya que el microprocesador funcionará siempre en modo maestro y la EEPROM en modo esclavo)

  De este modo cada vez que se active la interrupción se borrarán los *flags* de alarma y se actualizará una variable global (llamada “Estado” en los diagramas consiguientes) con el resultado de la última operación realizada.
- La función de espera activa: encargada de evaluar el resultado de la última operación efectuada sobre el bus I^2C (lectura o escritura) mediante *polling* (proceso de consulta reiterada) de la variable global de estado (que se actualiza mediante la rutina de atención de la interrupción I^2C) indicando después al exterior el éxito o fracaso de ésta. Simultáneamente activa un temporizador que si vence (llega a cero) sin recibir respuesta alguna del bus (mediante la llegada de interrupciones I^2C o cambios en el registro IICS), da la comunicación por fallida notificando el estado de error en el canal.

A continuación se presenta un diagrama simplificado de las dos funciones programadas para facilitar su comprensión:



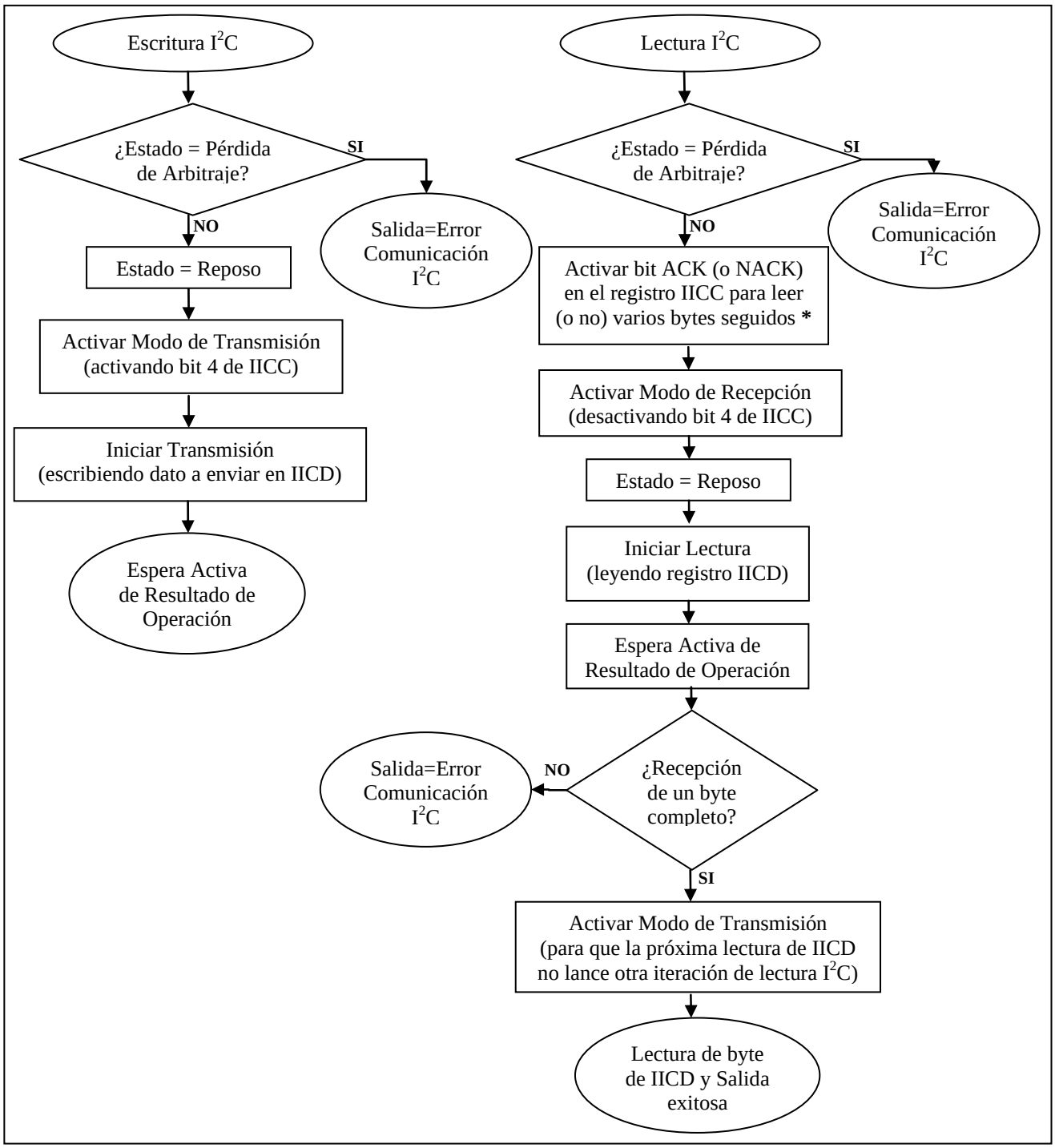
Espera activa de un ACK del bus y su relación con la interrupción I²C del MC13213

A continuación, basándose en las funciones de espera activa y de atención de interrupción I²C recién descritas, se desarrollaron las funciones principales de lectura y escritura I²C. Ambas serán después necesarias para realizar operaciones de lectura sobre la EEPROM auxiliar.

El algoritmo de lectura I²C programado inicializará el estado de los registros y los configurará para poner al MC13213 en modo recepción (dentro del rol de maestro) y para responder con un bit de ACK los bytes leídos del bus I²C. A continuación, se procederá a realizar una lectura del registro IICD que forzará el inicio de la operación de lectura sobre el bus. Finalmente, se espera (mediante las funciones de espera activa y de atención de interrupción I2C) a que el dispositivo esclavo termine de transmitir y se deposite exitosamente un nuevo dato para leer en el registro ICCD.

El algoritmo de escritura I²C programado inicializará el estado de los registros y los configurará para poner al MC13213 en modo transmisión (dentro del rol de maestro). A continuación, se realizará el envío de cada byte escribiendo el dato deseado en el registro IICD. Finalmente, se espera a recibir un asentimiento (ACK o NACK) o código de error del esclavo mediante la rutina de espera activa y de atención de interrupción ya explicadas.

A continuación se presenta un diagrama de las dos funciones descritas, que serán la base principal para las operaciones de lectura y escritura sobre la memoria auxiliar AT24C512 mediante el bus I^2C:



Proceso de lectura y escritura en el bus I^2C desde el MC13213

***:** El API I^2C diseñado soporta la lectura continua de bytes consecutivos del bus dentro de la misma comunicación (es decir, sin señales STOP intermedias). Para lograrlo bastará con activar la opción de ACK (en lugar de NACK) en el registro IICC para que una vez recibido cada byte del canal el MC13213 asienta la operación y continúe leyendo más bytes (mediante lecturas del registro IICD) repitiendo la secuencia para cada dato.

Llegados a este punto, se ha desarrollado un conjunto de funciones que permiten la comunicación entre la plataforma MC131213 y la memoria auxiliar AT24C512 a través de un protocolo conforme con el bus I^2C. No obstante, todavía es necesario emplear dichas funciones para la creación de un API de lectura (y posteriormente escritura) sobre la EEPROM que permita al usuario trabajar de forma transparente a dicho bus.

Así pues, basándose en las funciones I^2C recién desarrolladas y explicadas, a continuación se implementaron principalmente dos funciones de lectura y escritura sobre la EEPROM siguiendo los pasos descritos para ello en el apartado 3.9 acerca del bus I^2C y su aplicación para el componente AT24C512 (junto con su semántica específica para apuntar posiciones de memoria).

La función de lectura de la EEPROM comprende las siguientes actividades:

- Control de bus ocupado (mediante lectura de un bit concreto del registro IICS) y configuración de la comunicación (borrado de los *flags* del IICS, reinicialización de las variables de estado y selección del rol de maestro en el registro IICC)
- Envío de bit START
- Escritura I^2C de la dirección de la EEPROM y del bit R/W a 0 (escritura)
- Escritura I^2C de la posición de memoria a leer (2 bytes)
- Envío de bit de START repetido (mediante el registro IICC) para cambiar el modo de transmisión
- Escritura I^2C de la dirección de la EEPROM de nuevo pero ahora con el bit R/W a 1 (cambio a modo lectura)
- Lectura I^2C de uno o varios bytes de datos
- Finalización de la operación de lectura, limpiado de registros de estado y comprobación de errores.

La función de escritura de la EEPROM es más sencilla y comprende las siguientes actividades:

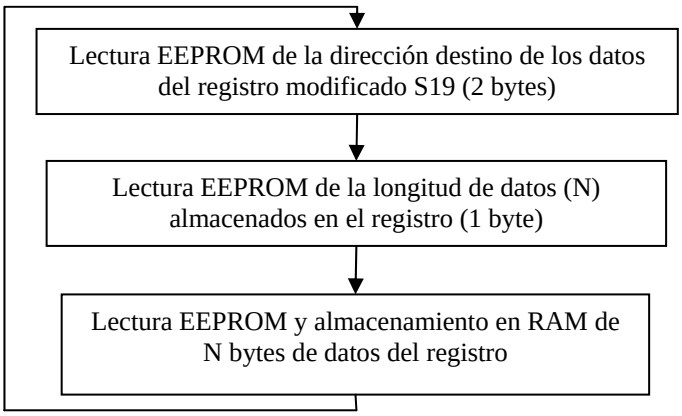
- Control de bus ocupado (mediante lectura de un bit concreto del registro IICS) y configuración de la comunicación (borrado de los *flags* del IICS, reinicialización de las variables de estado y selección del rol de maestro en el registro IICC)
- Envío de bit START
- Escritura I^2C de la dirección de la EEPROM y del bit R/W a 0 (escritura)
- Escritura I^2C de la posición de memoria sobre la que se va a empezar a escribir (2 bytes)
- Escritura I^2C de uno o varios bytes de datos
- Finalización de la operación de escritura, limpiado de registros de estado y comprobación de errores
- Espera activa de diez milisegundos (tiempo mínimo recomendado por el fabricante del dispositivo AT24C512 entre dos escrituras sobre la memoria)

Una vez se ha programado un API completo para la lectura y escritura de posiciones de memoria de la EEPROM externa, el bucle que recupera los registros modificados S19 almacenados en la memoria AT24C512 es conceptualmente muy sencillo de diseñar.

Como ya se ha mencionado en el apartado 4.1.3, en el mapa de memoria de la EEPROM se podrán separar los siguientes dos bloques:

- Cabecera (14 bytes)
- Registros modificados S19, cada uno con un campo de dirección destino (2 bytes), un campo de longitud de registro (1 byte) y de uno a 32 bytes de datos (en ese orden).

De este modo, el proceso de lectura de registros modificados S19 se iniciará a partir de la posición 0x000E de la memoria EEPROM (justo después de la cabecera) y cada iteración seguirá el esquema expuesto a continuación:



Algoritmo recursivo de lectura de registros S19 modificados

Una vez se dispone en RAM de los datos del registro S19 modificado, se llamará al bucle de sobrescritura de memoria Flash (descrito en el próximo sub apartado) que decidirá si esos datos se copian inmediatamente a la Flash, si es necesario previamente borrar una nueva página (si los datos van destinados a alguna página que todavía no había sido formateada por el bootloader), o si se descartan (si los datos pertenecen a regiones protegidas como el bootloader, la NVM o la última página).

Existe una excepción que se dará cuando alguno de los datos del registro leído esté destinado a actualizar una posición de memoria contenida en la última página de la Flash. En este caso el control del proceso de actualización pasará al bloque funcional encargado de borrar la página Flash número 127 (bloque explicado después del bucle de sobreescritura Flash) el cual, cuando concluya su cometido, llamará de nuevo al bloque de lectura de EEPROM para que termine de leer todos los registros S19 modificados que quedan almacenados del nuevo *firmware*.

4.1.4.1.2. Bucle de Sobrescritura de la Memoria Flash

Este bloque funcional del Bootloader será el encargado de procesar los datos de los registros modificados S19 según se vayan recuperando éstos de la EEPROM mediante el bucle principal de lectura a través del bus I^2C. Las llamadas a este bloque se realizarán cada vez que el bloque de lectura de EEPROM (explicado en el apartado anterior) deposite en memoria volátil (RAM) datos destinados a reescribir posiciones concretas de la memoria Flash del dispositivo.

Así pues las funciones principales de este bloque, según se presentan esquematizadamente en subapartados anteriores, pueden resumirse en los siguientes puntos:

- Comprobar que los datos del registro (cada uno de ellos) no van destinados a posiciones reservadas de memoria Flash (páginas de la NVM o del propio bootloader), y descartarlos en caso afirmativo. En principio un diseño inteligente del sistema de envío y almacenamiento de imágenes no daría nunca lugar a esta situación ya que para optimizar recursos no se enviarían ni almacenarían nunca los registros S19 relativos a posiciones de memoria que no se actualizarán.
- Comprobar si alguno de los datos de cada registro leído va destinado a una página de memoria que no se ha borrado previamente. En caso afirmativo el algoritmo irá realizando llamadas periódicas a las funciones de borrado de páginas Flash reiniciándolas según sea necesario. No obstante, las páginas de memoria que no vayan a ser actualizadas por no existir datos almacenados destinados a ellas, no se borrarán para maximizar la vida útil de la Flash y la velocidad de la actualización.
- En el caso de disponer por primera vez de datos destinados a la última página de Flash, se encargará su borrado previo a un bloque funcional específico (descrito más adelante) que adicionalmente reescribirá el vector de interrupción I^2C y permitirá a este bloque seguir después con su funcionamiento habitual.
- Si las anteriores comprobaciones son superadas exitosamente, el algoritmo procederá a regrabar cada byte de datos del registro modificado S19 almacenado temporalmente en memoria RAM mediante llamadas al API de grabación Flash previamente programado.

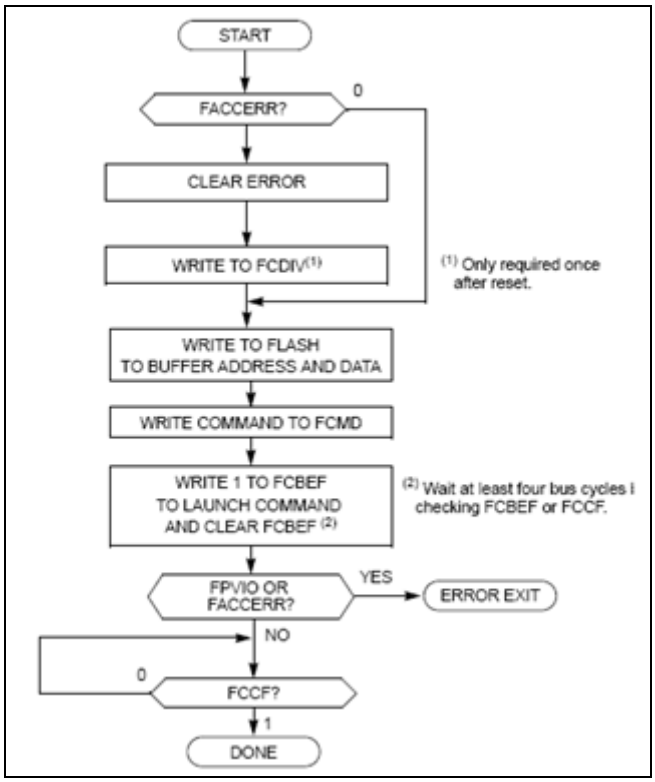
Así pues este bloque funcional, aparte de realizar una serie de comprobaciones sobre los datos depositados en RAM por el bloque de lectura de EEPROM, se encargará de la operación más delicada del proceso de actualización: el borrado y regrabado de la memoria Flash del dispositivo.

Para ello, la parte del código del bootloader encargada del borrado y regrabado de la memoria Flash se apoyará en el interfaz que suministra la plataforma MC13213 para el manejo de ésta y cuyo funcionamiento se ha explicado previamente en el apartado 4.1.2.2.

De este modo cuando se necesite realizar una operación de borrado o reescritura de la memoria Flash el bootloader seguirá, basándose en los registros **FCDIV**, **FSTAT** y **FCMD,** el siguiente algoritmo:

- Reiniciar los registros implicados (principalmente el registro **FCDIV**, que sólo se configurará una vez al principio del bootloader) y limpiar toda notificación de errores pasados.
- Escribir un byte de datos en una dirección de la memoria Flash (esta operación realmente no se ejecutará todavía, pero el acto de solicitarla provoca que el controlador Flash encole la petición). En el caso de que la operación que se vaya a solicitar a continuación no sea una escritura, el byte de datos "escrito" en la Flash será irrelevante, ya que sólo se empleará para indicar al interfaz el deseo de ejecutar un comando sobre una página determinada de la memoria no volátil (aquella que incluya la posición sobre la que se ha solicitado grabar el byte de datos).
- Escribir el comando deseado en el registro **FCMD** para encolar la petición en el buffer del controlador Flash. De los cinco comandos posibles, el bootloader solo empleará el de grabación estándar (0x20) o el de borrado de página (0x40) dependiendo de cada fase del proceso de actualización de la memoria (ver apartado 4.1.2.2 para una explicación detallada de las razones de este hecho).
- Escribir un 1 en el bit FCBEF del registro **FSTAT** para limpiar el *flag* y lanzar el procesado del comando solicitado.
- Comprobar el bit FCCF del registro **FSTAT** hasta que se active (a 1) indicando que la operación solicitada ha finalizado. A continuación se comprobará si ha tenido lugar algún error y se limpiarán los registros de estado para prepararlos para cualquier operación posterior.

Todos estos pasos pueden observarse de forma esquemática en el siguiente esquema aplicable tanto para las operaciones de regrabado como para las de borrado de página, cambiando sólo el valor del comando enviado al interfaz Flash en cada caso:



Algoritmo de escritura y borrado de la memoria flash [33]

Como ya se ha explicado en apartados anteriores, uno de los inconvenientes que plantea la memoria Flash de la plataforma MC13213 es que recomienda que no se continúe leyendo de la memoria ROM mientras se está procesando o solicitando un comando de borrado o reprogramación de la Flash. No acatar esta recomendación puede provocar errores de lectura que den lugar a problemas graves en el flujo natural del programa.

Este “grave” inconveniente se ha resuelto programando todo el algoritmo del esquema anterior directamente en ensamblador, de forma que cuando es necesario ejecutarlo el bootloader realiza una copia de éste a una posición de la memoria RAM junto con los datos a grabar (en el caso de ejecutarse una escritura en Flash). A continuación se apunta el contador de programa a dicha posición, de forma que el algoritmo continúa ejecutándose desde memoria volátil mientras dura el proceso de actualización de la Flash. Una vez termina el procesado del comando solicitado, el contador de programa vuelve a apuntar a la posición de memoria Flash original del bootloader.

Como puede apreciarse, el algoritmo más crítico del bootloader es considerablemente sencillo de implementar y el código resultante ocupará poca memoria (de lo contrario sería imposible depositarlo en RAM mientras se procesa cada comando sobre la memoria Flash).

No obstante, la problemática principal de este bloque reside en que se ejecutará muchas veces a lo largo de una actualización de *firmware* y que los errores que puedan darse durante este proceso tienen difícil o nula solución (la mayor parte de ellos derivados de una posible pérdida

de alimentación durante el proceso de borrado y regrabado). Es por ello que este bloque se ha programado optimizando su velocidad de procesado y reduciendo al máximo el tamaño del código final generado.

Una vez este bloque funcional ha terminado de procesar todos los datos que el bloque de lectura de la EEPROM había depositado para él en RAM (proceso que puede comprender descartar bytes, borrar páginas y/o regrabar posiciones de Flash), se devolverá el control al segundo para que continúe leyendo de la EEPROM datos destinados a la actualización de páginas de la memoria Flash (que seguramente volverán a resultar en nuevas llamadas a este bloque).

Cuando este bloque termine de reescribir las últimas posiciones de memoria Flash (0xFFFE y 0xFFFF) correspondientes al vector de interrupción de reset, significará que el proceso de borrado y regrabado de la memoria Flash está a punto de concluir y se cederá el control al bloque final de la actualización del *firmware* (explicado más adelante).

4.1.4.1.3. Bloque de Sobrescritura de la Página Flash 127

Como ya se ha mencionado, aunque casi la totalidad del proceso de actualización del *firmware* tendrá lugar en los dos bloques funcionales que se acaban de describir, el borrado de la última página de la Flash (la número 127) requerirá de un tratamiento especial que se desarrollará en este bloque funcional.

Esto se debe a que dicha página contiene los registros no volátiles, el puntero de inicio de programa y los vectores de interrupción, entre los cuales se encuentra el responsable de la comunicación I^2C, imprescindible para realizar lecturas sobre la memoria auxiliar EEPROM externa (ver apartado 4.1.2.4). De este modo, de realizarse un borrado de la última página y no sobrescribirse inmediatamente el valor del vector de interrupción I^2C, ya no podrían leerse nuevos registros S19 de la EEPROM ni terminar pues el proceso de actualización de la Flash.

Puesto que el valor almacenado en dicho vector de interrupción (concretamente en las posiciones 0xFFCE y 0xFFCF) no debería de cambiar nunca para no alterar el funcionamiento del bootloader, no será necesario recorrer los registros S19 en busca de dicho valor. Bastará con leer el valor antiguo almacenado en el vector de interrupción, borrar la página y reescribir dicha posición con el mismo valor.

La plataforma MC13213 recomienda no reescribir más de una vez una posición de memoria Flash después del borrado completo de la página que la contiene pero no impone ninguna restricción acerca del orden en que se han de reescribir las posiciones de cada página. Así

pues, el bootloader actualizará todas las páginas de la Flash para las que tiene datos almacenados reescribiendo sus bytes secuencialmente en orden creciente de dirección a excepción de la última página, que borrará completamente, reescribirá el vector de interrupción I^2C y finalmente el resto de datos (de nuevo secuencialmente en orden creciente e ignorando el valor repetido del vector I^2C).

Para realizar las operaciones de borrado de la página 127 y la sobreescritura del vector de interrupción I^2C por su mismo valor, este bloque funcional hará uso de las mismas funciones empleadas en el bloque funcional de sobreescritura de la Flash (explicadas justo en el subapartado anterior).

Una vez finalizado pues el cometido de este bloque, se devolverá el control al bucle de sobreescritura Flash para que termine de reescribir el resto de posiciones de memoria Flash de la página 127. Dicho bloque ignorará las posiciones de memoria asociadas al vector de interrupción I^2C para no incurrir en el error de volver a reescribirlas sin haber vuelto a borrar la página entera.

Cuando el bucle de sobreescritura reescriba las ultimas posiciones de memoria Flash, asociadas al vector de reset (0xFFFE y 0xFFFF), dará por terminado su trabajo y transferirá el control al bloque de comprobaciones finales y reset, que se describe a continuación.

4.1.4.1.4. Comprobaciones Finales y Reset (Fin de la Actualización)

En el momento en que el resto de bloques funcionales desarrollados en los subapartados anteriores han terminado su trabajo, se llama a este bloque final para concluir con el proceso de actualización del *firmware* del ND07.

Llegados a este punto, todos los errores que puedan haber tenido lugar a lo largo del proceso de actualización han debido de ser convenientemente solucionados mediante las técnicas que se describen en el apartado siguiente 4.1.4.2.

Si el proceso de actualización ha considerado conveniente otorgar el control a este último bloque funcional del bootloader significa que da la sustitución del *firmware* por finalizada satisfactoriamente.

No obstante, una vez alcanzado el bloque de comprobaciones finales, aún resta modificar el *flag* de regrabación de la EEPROM para indicar que la actualización ha finalizado y que la próxima reinicialización del dispositivo ha de arrancar el nuevo *firmware* ignorando la imagen almacenada en la memoria externa.

La actualización del valor del *flag* de regrabación se realizará empleando el API desarrollado para la lectura y escritura de la EEPROM explicado en la sección “Bucle de Lectura de Registros Modificados S19” de este mismo subapartado.

Así pues, este bloque se encargará de reescribir el *flag* de regrabación de la EEPROM (posiciones de memoria 0x0000 y 0x0001, ver apartado 4.1.3) a un valor **0xCCCC**. Obviamente esta grabación también podría dar errores y es competencia del algoritmo reintentarla hasta tener éxito. En caso de superar los 255 reintentos se dejaría de reintentar y se confiaría que el próximo rearranque del dispositivo (que obviamente procederá, al seguir el *flag* de regrabación activo, a volver a actualizar la memoria Flash) arreglará el problema.

Lo más habitual será que de fábrica el valor de las posiciones de memoria de la EEPROM sean todo bytes a 0x00 o 0xFF. Es por ello que no resulta conveniente emplear el valor 0x0000 para indicar que la imagen almacenada ha sido empleada exitosamente para actualizar el dispositivo (pero que no se desea realizar de nuevo de momento) y se emplea en su lugar **0xCCCC**.

La aplicación encargada de almacenar *firmwares* en la EEPROM podrá emplear el *flag* de regrabación para indicar en cualquier momento el estado de una imagen en la memoria externa, como por ejemplo: en proceso de recepción, lista para ser redistribuida, lista para actualizar el dispositivo anfitrión (con el valor **0xAABB** como ya se ha comentado), obsoleta, con errores, etc. La decisión acerca del significado de cada valor (a excepción de los valores reservados 0xAABB, 0xCCCC, 0xFFFF y 0x0000) queda en manos del futuro desarrollador de la aplicación encargada de almacenar las imágenes en EEPROM.

Una vez el *flag* de regrabación se ha desactivado exitosamente para no volver a activar el bootloader en el próximo arranque del dispositivo, se procederá a resetear el ND07 para que comience a ejecutar el nuevo *firmware*.

Para lograrlo, se ejecutará una instrucción fuera del rango permitido del MC13213, que provocará la interrupción inmediata del flujo del programa y reiniciará el dispositivo de igual forma que si hubiese perdido la alimentación. A continuación, se ejecutará el nuevo cargador de arranque que pasará por el bootloader para comprobar que no existe otra imagen disponible en EEPROM para actualizar el *firmware*. Después se otorgará el control al resto de la aplicación “actualizada”.

Llegados a este punto se considera finalizada la descripción “modular” del funcionamiento del bootloader desarrollado, objetivo final de este proyecto.

4.1.4.2. Validación de la Actualización del *Firmware*

La mayor parte de las incidencias que se pueden dar durante la ejecución del algoritmo de actualización se pueden separar fácilmente en errores de grabación/borrado de posiciones de memoria Flash o en errores de lectura/escritura de la EEPROM a través de la comunicación I^2C.

Desafortunadamente, en ambos casos la estrategia empleada para "solucionar" el error se basa en el reintento de la operación hasta que funcione o se descarte (tras reintentar un máximo de 255 veces) dando por hecho que la propia notificación de error no es fiable. Esto es así porque, como ya se explicó en el apartado 4.1.1, el objetivo de este proyecto es únicamente el desarrollo del bootloader encargado de la actualización del *firmware*, de modo que éste no puede contar con la disponibilidad de un cargador de arranque completo protegido que permanezca inmutable y que permita descartar la actualización en caso de darse un problema irrecuperable a mitad del proceso.

Es necesario recordar que existe un límite máximo en el número de ciclos de escritura sobre la memoria Flash y sobre la EEPROM, de modo que reintentar una operación demasiadas veces dañaría el dispositivo permanentemente.

La notificación hacia el exterior de la aparición de un error en el proceso de actualización es de poca o nula utilidad ya que si el éste se deja a medias para tratar de informar al usuario, probablemente el dispositivo no vuelva a funcionar correctamente nunca más. Es por ello que la política de actualización empleada para dispositivos de poca memoria y capacidad de procesamiento como el ND07 consiste básicamente en reintentar las operaciones que dan errores y no notificar nada al exterior. Además, en caso de la aparición de un error de grabación (irrecuperable) el usuario poco podría hacer para arreglarlo más allá de sustituir el dispositivo por otro.

Así pues, el código del bootloader comprueba el resultado de la mayor parte de operaciones que va realizando (en ocasiones, para simplificar el código, se realizan comprobaciones de algoritmos completos) y cuando éstas no finalizan exitosamente se reintentan tras reinicializar el estado de los registros de estado.

El caso más desafortunado es un fallo en la escritura de una posición de memoria Flash. Cuando esto sucede es imprescindible volver a borrar la página y repetir la actualización de esta de cero, lo cual implica volver a leer registros S19 modificados ya procesados, con el coste computacional que supone. Una vez más, cada página se reintenta un máximo de 255 veces, tras lo cual se pasa a la siguiente confiando en el éxito de la operación.

La aparición constante de errores de grabación de posiciones Flash indicará claramente un defecto en la memoria de la plataforma MC13213 o un problema en la alimentación del dispositivo (el ND07 no dispone de batería de *backup*). Situaciones ambas de difícil solución: sustituir el ND07 o el adaptador de corriente por otro nuevo, respectivamente.

Adicionalmente, fluctuaciones de la tensión de alimentación pueden causar también errores en el proceso de grabación que sean indetectables por el controlador Flash, de forma que el bootloader podría llegar a dar por buena una actualización no exitosa. Sin embargo, la solución de contingencia para este problema pasa por comparar byte a byte la memoria Flash con la imagen almacenada en EEPROM o implementar algún tipo de *checksum* global.

Desafortunadamente ambos procesos pueden acarrear nuevos errores y complicarían demasiado el código sin tener a cambio la garantía de corregir el error (pudiendo intercambiarlo por otro). Es necesario elegir un punto de equilibrio entre complejidad (mayor lentitud) y seguridad del bootloader, y es por ello que estas soluciones de contingencia no se han implementado.

Para finalizar, ya se ha comentado que la validación de la actualización de *firmware* se realizará a lo largo de la ejecución del bootloader en cada operación que se realice, de forma que el algoritmo es bastante seguro. No obstante, una práctica recomendable para reducir al máximo las posibilidades de aparición de errores será exigir a la aplicación que ejecuta el bootloader que realice todas las comprobaciones de integridad posibles a la imagen depositada en EEPROM y que no permita el inicio de actualizaciones si se detectan fluctuaciones en la alimentación.

Todas estas recomendaciones se recogen en el apartado de requisitos para la aplicación del apéndice A.2.

4.1.4.3. Opciones del Compilador y del Enlazador (*Linker*).

El IDE *CodeWarrior for MicroControllers v5.1* incluye en su paquete un compilador, un ensamblador y un enlazador (*Linker*) capaces de generar código máquina para chips como el MC13123. Adicionalmente provee de un depurador que permitió en su momento limpiar el código programado, detectar errores y validar su correcto funcionamiento.

Tanto el compilador como el enlazador son configurables mediante parámetros de entrada, lo cual les convierte en herramientas muy flexibles y potentes a la hora de controlar el modo en que se generará el código desarrollado y se dispondrá en las memorias RAM y Flash del chip.

Independientemente de las optimizaciones de compilación seleccionadas, el compilador tratará siempre de optimizar la velocidad y tamaño del código generado, creando dependencias entre las funciones programadas y otras partes del código o sus propias librerías internas para así tratar de eliminar las redundancias encontradas al máximo.

Desafortunadamente, esta potencia de optimización del código por parte del compilador puede llegar a ser un problema para el desarrollo de una aplicación como el bootloader, cuyo requisito principal es que resida en una sección concreta de la memoria (tanto Flash como RAM) y no dependa en ningún momento de código o memoria alojada fuera de su propio ámbito. Esto se debe a que el proceso de actualización reescribirá toda la memoria del chip a excepción de aquella ocupada por el bootloader y, si éste dependiese de funciones alojadas fuera de su ámbito, dejaría de funcionar a mitad del proceso dejando al dispositivo en un estado irrecuperable.

Como ya se explicará más adelante, no existe opción seleccionable de compilación que fuerce al compilador a no optimizar una sección del código (como sería el caso del bootloader) de forma que siempre sea independiente del resto de la aplicación. Habitualmente sucede que a una aplicación cuyo bootloader es independiente del resto del código se le modifica alguna rutina de la comunicación ZigBee y el nuevo código generado convierte al bootloader en dependiente de la aplicación principal.

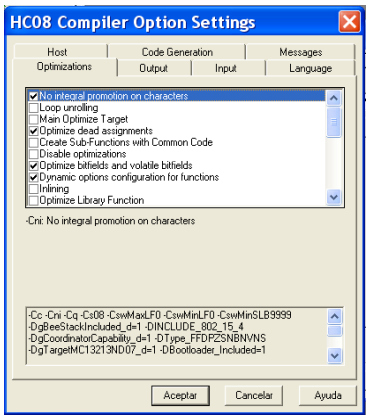
Sin embargo, la elección de una serie de optimizaciones del compilador frente a otras sí que puede provocar que en la mayoría de las aplicaciones compiladas el bootloader permanezca independiente al resto del código e inalterado, aunque sin garantizar nunca el 100% de los casos (dato que confirmó más tarde el servicio técnico de *Freescale*, ver apartado 4.1.5).

Es por todo esto que el estudio de las opciones del compilador y del enlazador ha sido fundamental para alcanzar el objetivo final de este proyecto (la obtención de un bootloader independiente del resto de código de aplicación) y se describen resumidamente a continuación.

4.1.4.3.1. Opciones del Compilador

Las opciones del compilador suministrado por *Freescale* dentro del IDE *CodeWarrior for Microcontrollers V5.1* se encuentran accesibles para su configuración siguiendo la ruta: *Edit -> Application Settings->Compiler for HC08* y pulsando el botón etiquetado como "*Options*".

Tras pulsar dicho botón (aunque las opciones pueden configurarse a mano en la entrada de texto etiquetada como "*Command Line Options*" de la pantalla que lo contiene), aparece el siguiente menú compuesto de pestañas y *checkboxes*:



Captura de pantalla del menú de opciones del compilador

Como puede apreciarse en la imagen anterior, las diversas optimizaciones aplicables por el compilador son accesibles de forma muy intuitiva y documentada (aunque sus efectos son en ocasiones más difíciles de apreciar) y las más importantes pueden resumirse en el siguiente listado junto con una breve descripción de su funcionamiento en inglés:

-I: filenames to DOS length
-AddIncl: Additional Include File
-Ansi: Strict ANSI
-Asr: It is assumed that HLI code saves written registers
-BfaB: Bitfield Byte Allocation
-BfaGapLimitBits: Bitfield Gap Limit
-BfaTSR: Bitfield Type Size Reduction
-Cc: Allocate Constant Objects into ROM
-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers
-Ci: Tri- and Bigraph Support
-Cni: No Integral Promotion
-Cppc: C++ Comments in ANSI-C
-Cq: Propagate const and volatile qualifiers for structs
-Cs08: Generate Code for HCS08
-CswMaxLF: Maximum Load Factor for Switch Tables
-CswMinLB: Minimum Number of Labels for Switch Tables
-CswMinLF: Minimum Load Factor for Switch Tables
-CswMinSLB: Minimum Number of Labels for Search Switch Tables
-Cu: Loop Unrolling
-Cx: No Code Generation
-D: Macro Definition
-Ec: Conversion from 'const T*' to 'T*'
-Eencrypt: Encrypt Files
-Ekey: Encryption Key
-Env: Set Environment Variable
-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format
-Fd: Doubles are IEEE32
-H: Short Help
-I: Include File Path
-La: Generate Assembler Include File
-Lasm: Generate Listing File
-Lasmc: Configure Listing File
-Ldf: Log Predefined Defines to File
-Li: List of Included Files
-Lic: License Information
-LicA: License Information about every Feature in Directory
-OnX: Disable Frame Pointer Optimization
-Or: Allocate Local Variables into Registers
-Ou and -Onu: Optimize Dead Assignments
-Pe: Preprocessing Escape Sequences in Strings
-Pio: Include Files Only Once
-Prod: Specify Project File at Startup
-Qvtp: Qualifier for Virtual Table Pointers
-Rp (-Rpe, -Rpt): Large Return Value Type
-T: Flexible Type Management
-V: Prints the Compiler Version
-View: Application Standard Occurrence
-WErrFile: Create "err.log" Error File
-Wmsg8x3: Cut filenames in Microsoft Format to 8.3
-WmsgCE: RGB Color for Error Messages
-WmsgCF: RGB Color for Fatal Messages
-WmsgCI: RGB Color for Information Messages
-WmsgCU: RGB Color for User Messages
-WmsgCW: RGB Color for Warning Messages
-LicBorrow: Borrow License Feature
-LicWait: Wait until Floating License is Available from Floating License Server
-LI: Statistics about Each Function
-Lm: List of Included Files in Make Format
-LmCfg: Configuration of list of Included files in Make Format
-Lo: Object File List
-Lp: Preprocessor Output
-LpCfg: Preprocessor Output configuration
-LpX: Stop after Preprocessor
-M (-Ms, -Mt): Memory Model
-N: Display Notify Box
-NoBeep: No Beep in Case of an Error
-NoDebugInfo: Do not Generate Debug Information
-NoEnv: Do not Use Environment
-NoPath: Strip Path Info
-O (-Os, -Ot): Main Optimization Target
-Obfv: Optimize Bitfields and Volatile Bitfields
-ObjN: Object filename Specification
-Oc: Common Subexpression Elimination (CSE)
-OdocF: Dynamic Option Configuration for Functions
-Of and-Onf: Create Sub-Functions with Common Code
-Oi: Inlining
-Oilib: Optimize Library Functions
-Ol: Try to Keep Loop Induction Variables in Registers
-Ona: Disable Alias Checking
-OnB: Disable Branch Optimizer
-Onbf: Disable Optimize Bitfields
-Onbt: Disable ICG Level Branch Tail Merging
-Onca: Disable any Constant Folding
-Oncn: Disable Constant Folding in case of a New Constant
-OnCopyDown: Do Generate Copy Down Information for Zero Values
-OnCstVar: Disable CONST Variable by Constant Replacement
-One: Disable any Low-level Common Subexpression Elimination
-OnP: Disable Peephole Optimization
-OnPMNC: Disable Code Generation for NULL Pointer to Member Check
-Ont: Disable Tree Optimizer
-OnX: Disable Frame Pointer Optimization
-Or: Allocate Local Variables into Registers
-Ou and -Onu: Optimize Dead Assignments
-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode
-WmsgFob: Message Format for Batch Mode
-WmsgFoi: Message Format for Interactive Mode
-WmsgFonf: Message Format for no File Information
-WmsgFonp: Message Format for no Position Information
-WmsgNe: Number of Error Messages
-WmsgNi: Number of Information Messages
-WmsgNu: Disable User Messages
-WmsgNw: Number of Warning Messages
-WmsgSd: Setting a Message to Disable
-WmsgSe: Setting a Message to Error
-WmsgSi: Setting a Message to Information
-WmsgSw: Setting a Message to Warning
-WOutFile: Create Error Listing File
-Wpd: Error for Implicit Parameter Declaration
-WStdout: Write to Standard Output
-W1: No Information Messages
-W2: No Information and Warning Messages

Listado de opciones de compilación disponibles [45]

Dependiendo del código fuente a compilar, cada optimización del compilador presenta una serie de ventajas y de desventajas. A pesar de ello, *Freescale* recomienda emplear siempre las siguientes opciones [45]:

- Alojar objetos constantes en ROM (opción **–Cc**) especificando su localización mediante la sección ROM_VAR en el fichero de parámetros del enlazador (*linker*).
- Notificar error cuando se encuentren declaraciones de parámetros implícitos (opción **-Wpd**), ayudando a detectar fallos de programación.
- Realizar optimización de registros (opción **–Or**) siempre que sea posible.
- Compilar las funciones de cada módulo con las opciones más adecuadas para él. Para ello se definirán dinámicamente mediante la opción **–OdocF** (*Dynamic Option Configuration for Functions*).

Las opciones por defecto del CodeWarrior activan la mayor parte de las optimizaciones del compilador. Esto puede provocar que, en ocasiones, el código resultante sea difícil de depurar, ocupe más de lo debido (cuando el objetivo a optimizar no sea el tamaño del código pero sí la velocidad) o incluso no funcione bien. Será función del programador estudiar pues cuales son las optimizaciones más adecuadas para la aplicación que desarrolla.

Así pues, a lo largo del desarrollo del bootloader se ha comprobado imprescindible omitir la opción **–Os** (es decir, forzar la optimización del tamaño del código frente a la velocidad), de lo contrario el código resultante siempre era dependiente de la aplicación principal. Por otro lado, el empleo de la opción **–Ot** (opción opuesta, en la que el compilador optimiza la velocidad del código) no genera dependencias del bootloader a priori pero aumenta considerablemente el tamaño del código resultante (factor que tampoco interesa). De forma que se descartaron ambas.

Las opciones de compilación finalmente empleadas para el Cargador RS232 con bootloader que no generaban dependencias fueron las siguientes:

```
-Cc –Cni –Cq –Cs08 –CswMaxLF0 –CswMinLF0 –CswMinSLB9999 –
DgBeeStackIncluded_d=1
-DINCLUDE_802_15_4 –DgCoordinatorCapability_d=1 –Dtype_FFDPZSNBNVNS
-DgTargetMC13213ND07_d=1 –DgMeshNetworkingCapability_d=1 –DgZtcIncluded_d=0
-Lasm=%n.lst –Lasmc=h –Ms –Ou –Obfv –OdocF=-Onca|-One|-Or|-Ont –OnB –TE1uE
-WmsgNu=acdet –Wpd
```

4.1.4.3.2. Pragmas

Un pragma es una directiva del compilador que define cómo se pasará la información desde el interfaz del programador hacia la parte encargada de procesar el código, sin afectar al analizador sintáctico (*parser*).

Los pragmas aceptados por el compilador suministrado por *Freescale* para la plataforma MC13213 se enumeran a continuación:

```
#pragma CODE_SEG: Code Segment Definition
#pragma CONST_SEG: Constant Data Segment Definition
#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing
#pragma DATA_SEG: Data Segment Definition
#pragma INLINE: Inline Next Function Definition
#pragma INTO_ROM: Put Next Variable Definition into ROM
#pragma LINK_INFO: Pass Information to the Linker
#pragma LOOP_UNROLL: Force Loop Unrolling
#pragma mark: Entry in CodeWarrior IDE Function List
#pragma MESSAGE: Message Setting
#pragma NO_ENTRY: No Entry Code
#pragma NO_EXIT: No Exit Code
#pragma NO_FRAME: No Frame Code
#pragma NO_INLINE: Do not Inline Next Function Definition
#pragma NO_LOOP_UNROLL: Disable Loop Unrolling
#pragma NO_RETURN: No Return Instruction
#pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing
#pragma ONCE: Include Once
#pragma OPTION: Additional Options
#pragma STRING_SEG: String Segment Definition
#pragma TEST_CODE: Check Generated Code
#pragma TRAP_PROC: Mark function as interrupt Function
```

Listado de PRAGMAS del compilador [45]

A efectos del desarrollo del bootloader, los pragmas CODE_SEG, CONST_SEG, DATA_SEG, han sido fundamentales para tener control preciso sobre las secciones concretas de memoria (tanto ROM como RAM) que requerirá la aplicación para su correcto funcionamiento.

Adicionalmente se ha empleado profusamente el pragma MESSAGE (para controlar las alarmas generadas en tiempo de compilación) y puntualmente el pragma OPTION (para especificar un tratamiento especial de una sección concreta del código por parte del compilador).

4.1.4.3.3. Opciones del Enlazador y Fichero de Parámetros(PRM)

Un enlazador (*linker*) es un programa que toma los ficheros de código objeto generados en los primeros pasos del proceso de compilación, la información de todos los recursos necesarios (librerías), quita aquellos recursos que no necesita, y enlaza el código objeto con su(s) librería(s) con lo que finalmente produce un fichero ejecutable o una librería.

Aunque el enlazador permite configurar distintas opciones (desde línea de comandos) de enlazado de los ficheros objeto generados en la compilación, se suele optar por aquellas que *Freescale* suministra por defecto para proyectos diseñados para la plataforma MC13213.
Estas opciones serán las siguientes:

```
-B –M –WmsgSd1100 –WmsgSd1912 –EnvSRECORD=s19 –WstdoutOn
```

Se ha comprobado que la elección de las opciones de enlazado por defecto dan lugar a compilaciones de aplicaciones independientes del bootloader siempre y cuando se hayan elegido correctamente las optimizaciones de compilación.

No obstante, debido a su importancia en el desarrollo exitoso de este proyecto, cabe destacar las siguientes opciones de enlazado:

- **-B**: opción que activa la generación de un fichero .S19 junto con el código compilado (no sólo código máquina). Esto permitirá a las aplicaciones enviar imágenes de *firmware* con un formato cómodo a través del puerto serie o de forma inalámbrica.
- **-M:** opción que permite generar mapas de memoria del código compilado, enlazado y ensamblado. Gracias a estos mapas se ha podido evaluar cuando una compilación concreta generaba dependencias entre la aplicación principal y el bootloader y detectar en qué partes concretas del código sucedía.

Sin embargo, la parte más importante del enlazador a efectos del bootloader es su fichero de parámetros.

El fichero de parámetros, de extensión **.prm** y sito en la carpeta homónima de cada proyecto del *CodeWarrior for MicroControllers v5.1*, permite al desarrollador etiquetar secciones distintas de la memoria Flash y RAM para luego emplazar cada bloque del código de aplicación (y lo más importante, del bootloader) en la sección que se desee.

El formato del fichero de parámetros **.prm** es un fichero de texto en el que se distinguen principalmente los siguientes bloques: [46]

- bloque **SECTIONS**: en el que se separa la memoria en varias secciones, especificando el tipo de datos que van a alojar (de sólo lectura, lectura y escritura, etc.) y otorgándoles una etiqueta.
- bloque **PLACEMENT**: en el que se asigna a cada segmento de la memoria (definidos a lo largo del código mediante los pragmas CODE_SEG, CONST_SEG o DATA_SEG y un identificador específico) a una sección de la memoria de las anteriormente definidas.
- Bloque **VECTOR 0**: vector que apunta a la primera función que se ejecutará al resetear el dispositivo.

Aunque se podrían mencionar más tipos de bloques y explicar extensamente las secciones de memoria predefinidas existentes, no son directamente relevantes en cuanto al bootloader se refiere. No obstante, un desarrollador de una aplicación ZigBee estándar sí que deberá estudiar este fichero con detenimiento para asegurarse de que todo su código se emplaza donde tiene planeado.

El bootloader, como se explicará mas adelante, deberá reservar una sección de memoria Flash y RAM para su uso particular que no podrá ser empleada para ninguna otra aplicación, o de lo contrario no se podrá asegurar su correcto funcionamiento el 100% de las veces.

Para ello será necesario pues incluir en **el fichero de parámetros del enlazador,** en todas las aplicaciones que incluyan el bootloader desarrollado en este proyecto, las siguientes secciones (dentro del bloque **SECTIONS**):

```
//RAM volátil reservada por el Bootloader
BOOTLOADER_RAM_SECTION        = READ_WRITE 0x0260 TO 0x02A4;

//Vector de interrupción I2C para aplicación (siempre justo antes del bootloader)
I2C_VECTOR_SECTION      = READ_ONLY  0xF7FE TO 0xF7FF;

//Secciones BOOTLOADER
NLAZA_BOOT                    = READ_ONLY  0xF800 TO 0xF85F;   //0xFDFF;
NLAZA_CODE                    = READ_ONLY  0xF860 TO 0xFDFF;   //0xFDFF;
```

Y las siguientes asignaciones de memoria (dentro del bloque **PLACEMENT**):

```
//BOOTLOADER
NLAZA_BOOTLOADER        INTO NLAZA_BOOT;
NLAZA_BOOTCODE          INTO NLAZA_CODE;
BOOTLOADER_RAM          INTO BOOTLOADER_RAM_SECTION;

//Código Interrupción I2C para aplicación
I2C_APP_VECTOR          INTO I2C_VECTOR_SECTION;
```

De este modo, cuando un programador de dispositivos ZigBee ND07 necesite incluir el bootloader en su código, deberá incluir estas líneas adicionales en el fichero de parámetros (y modificar el resto en consecuencia para no solapar secciones) a la vez que incluye los ficheros del bootloader en el proyecto de su aplicación.

4.1.4.4. Programar Código en Ensamblador

Como se deduce de los apartados anteriores, el principal problema que planteó el bootloader, una vez diseñado y programado, fue que dependiendo de las opciones de compilación elegidas (y en ocasiones sin importar que optimizaciones se aplicasen) el código de éste se hacía dependiente de otras secciones de la memoria, externas a su rango asignado.

Seleccionar unas opciones de compilación frente a otras eliminaba en ocasiones las dependencias generadas pero obviamente alteraba el modo en que se ensamblaba el bootloader, modificando el tamaño del código resultante.

Puesto que el bootloader reserva 3 páginas de memoria para almacenar su código de aplicación y habitualmente (en la mayoría de las compilaciones) la última se queda casi vacía, ligeras modificaciones del tamaño no tienen un efecto negativo significativo (siempre y cuando no se altere su funcionamiento).

No sucede así con las dependencias. En el momento que el bootloader se convierte en dependiente de otra sección del código ya no se puede asegurar su correcto funcionamiento. Afortunadamente el enlazador (*linker*) permite generar mapas de memoria del código compilado (ver apartado anterior) de forma que se puede detectar rápidamente cualquier dependencia, así como evaluar el tamaño del código resultante y comprobar si se ha almacenado en la sección que tiene asignada en el fichero de parámetros.

Aunque podría llegar a ser una restricción asumible en un momento dado, en un principio los esfuerzos de desarrollo del bootloader han ido siempre encaminados a hacerlo funcional de forma independiente a las optimizaciones del compilador elegidas por el programador, para evitar restar flexibilidad al diseño del resto de aplicaciones.

Finalmente, se llegó a la conclusión de que el único método que garantiza que el bootloader se compilará de forma independiente al resto de la aplicación principal entre las sucesivas versiones de ésta, consiste en desarrollarlo íntegramente en ensamblador y luego grabarlo en la posición deseada de memoria Flash (cuando se programe el ND07 por primera vez para sacarlo al mercado). Adicionalmente este método aseguraría un tamaño y posición en memoria del bootloader fijas sea cual sea la aplicación principal que lo contenga.

Esta conclusión fue confirmada tiempo más tarde por el servicio técnico de *Freescale* (ver apartado 4.1.5) tras realizar muchas consultas.

Desafortunadamente, programar el bootloader íntegramente en ensamblador es extremadamente costoso a efectos de tiempo y trabajo, de forma que la primera aproximación fue la de traducir a ensamblador sólo aquellas funciones del bootloader más propensas a ser optimizadas por el compilador generando dependencias con código externo al bootloader.

Una vez más, esta solución no resolvía el problema en la totalidad de las situaciones (por no hablar del hecho de que en muchos casos el compilador no trataba correctamente funciones con muchos parámetros de entrada cuando estaban íntegramente escritas en ensamblador) de forma que no quedó más remedio que traducir toda la aplicación a ensamblador y código máquina.

Para facilitar el trabajo en la medida de lo posible y minimizar el tiempo de desarrollo, el proceso de “traducción” se realizó en las siguientes fases:

- Fase I: Finalización del código definitivo (en C con secciones puntuales en ensamblador). Finalización del código definitivo del cargador RS232. Depuración de códigos hasta obtener la versión definitiva y comprobación de su funcionamiento.
- Fase II: Evaluación de las opciones de compilación hasta obtener una combinación que no generaba dependencias entre el código del bootloader y el de la aplicación principal (en este caso el cargador RS232). Reserva de secciones de memoria RAM y Flash para el bootloader mediante el fichero de parámetros del enlazador y el empleo de pragmas (ver apartado 4.1.4.3). Comprobación de correcto funcionamiento.
- Fase III: Apoyándose en los ficheros de listado generados en la compilación (mediante determinadas opciones del compilador) y en el depurador del CodeWarrior, se pudo recuperar una versión en ensamblador más o menos precisa de la última versión sin dependencias del bootloader sin necesidad de programarla de cero. A partir de esta primera versión en ensamblador se desarrolló la versión final con una pequeña sección en código C (encargada de asegurarse de que el CodeWarrior reservaría siempre la memoria necesaria en RAM y ROM para el bootloader), una pequeña sección en ensamblador (encargada de inicializar el módulo I^2C) y el resto de la aplicación en código máquina (que resultará inalterable sea cual sea la optimización del compilador elegida).

De este modo, al apoyarse en los ficheros de listados en ensamblador generados por el compilador, el proceso de migración del código en lenguaje C a código máquina y ensamblador se realizó de forma óptima y controlada, empleando una cantidad de tiempo mucho menor de lo que habría implicado programarlo de cero y obteniendo un resultado mucho más robusto.

A partir de este punto, todas las aplicaciones que incluyan el bootloader obtenido (traducido a código máquina a partir de una compilación en la que no se habían generado dependencias con la aplicación principal), podrán ser compiladas con cualquier conjunto de optimizaciones sin que se generen dependencias entre ambos puesto que éste ya se presenta compilado en código máquina y ensamblador como una ristra de bytes (no optimizable en modo alguno).

4.1.4.5. Localización Inteligente del Bootloader

Debido al hecho de que la versión definitiva del bootloader desarrollado a lo largo de este proyecto ha de suministrarse en ensamblador y código máquina (para evitar dependencias con el resto del código, como ya se ha comentado con anterioridad), la mayor parte de los parámetros que rigen su funcionamiento han de ser fijados con anterioridad a dicha “traducción”.

A continuación se desarrollan pues las decisiones de diseño tomadas referentes a la reserva de memoria y emplazamiento dentro de la secuencia de arranque del dispositivo. Éstas se convertirán en parte de los requisitos de instalación del bootloader dentro de una aplicación ZigBee.

4.1.4.5.1. Emplazamiento en la Memoria (RAM y Flash)

La elección y reserva de las posiciones de memoria Flash del MC13213 que ocuparán las tres páginas del bootloader es fundamental para el óptimo aprovechamiento de ésta y para evitar posibles problemas derivados de su solapamiento con secciones del código de la aplicación principal.
Puesto que el código relativo al bootloader no se sobrescribirá nunca, ocupará un número entero de páginas Flash que jamás deberán contener nada del resto de código de aplicación ni del espacio reservado para registros no volátiles o para vectores de interrupción del MC13213. De lo contrario, estos módulos (o secciones de código) no se sobrescribirían entre actualizaciones pudiendo corromper la aplicación final.

Si se analiza el modelo de memoria del MC13213 (ver apartado 4.1.2.2), se deduce que las tres páginas del bootloader (1535 bytes) podrían emplazarse o en el bloque de direcciones **[0x1080 → 0x1800]** o en el bloque **[0x192C → 0xFDFF]**. La última página, cuyo rango de direcciones es **[0xFE00 →0xFFFF]**, no puede emplearse para este fin ya que contiene los vectores de interrupción, de reset y los registros no volátiles del MC13213.

Sin embargo, puesto que por defecto el puntero de inicio de programa apunta siempre a la primera posición de Flash (**0x1080**) como emplazamiento del código de inicio del dispositivo (de tamaño no despreciable), las páginas asociadas al bloque de direcciones **[0x1080→0x1800]** no parecen ser las más adecuadas para la instalación del bootloader.

Adicionalmente, el módulo NVM suele reservar 3 páginas de memoria Flash para el almacenamiento de variables no volátiles que por defecto cubren el rango de direcciones **[0x1A00→0x1FFF]**. Puesto que el número de páginas empleadas por el módulo NVM puede ser mayor que 3 (aunque en esas situaciones, de suceder una actualización se sobrescribiría el valor de éstas), no resulta conveniente emplazar el bootloader inmediatamente después de las páginas reservadas para la NVM.

De este modo, se ha decidido emplazar el código del bootloader en las **páginas 124, 125 y 126** (rango de direcciones **[0xF800→0xFDFF]**) al final de la memoria Flash del dispositivo (evitando la última página, para que cualquier actualización de *firmware* pueda modificar el valor de un registro no volátil o de un vector de interrupción). Esta decisión de diseño deja al compilador,

ensamblador y enlazador (*linker*) con la mayor cantidad de memoria Flash libre contigua dentro del bloque **[0x192C → 0xFFFF]** posible y se protege ante las configuraciones atípicas que un desarrollador de aplicaciones podría decidir programar en una aplicación concreta.

En cuanto a consumo de RAM se refiere, el bootloader desarrollado necesita reservar una serie de variables y código de aplicación en memoria volátil que ocuparán 69 bytes de RAM de forma permanente.

Una vez más, se decide reservar la sección de memoria RAM con menor riesgo de colisión con otros módulos del microcontrolador. Con este fin, el bootloader reservará los primeros 69 bytes de la memoria RAM designada como "libre" después del stack. Así pues, cuando se desee instalar el bootloader, será necesario reservar el rango de direcciones **[0x0260 →0x02A4]** para su uso exclusivo.

Como ya se ha explicado en el apartado 4.1.4.3, el fichero de parámetros del enlazador (***.prm***) (presente en todo proyecto de *CodeWarrior*) permitirá al programador de aplicaciones reservar los rangos de memoria RAM y Flash especificados anteriormente para la correcta instalación del bootloader. Los pasos concretos a dar se enumerarán en el apéndice de instrucciones de instalación, en el apartado A.1.1.

4.1.4.5.2. Emplazamiento en la Secuencia de Arranque

La elección del emplazamiento del bootloader dentro de la secuencia de arranque del dispositivo es un proceso delicado ya que de él dependen muchos factores.

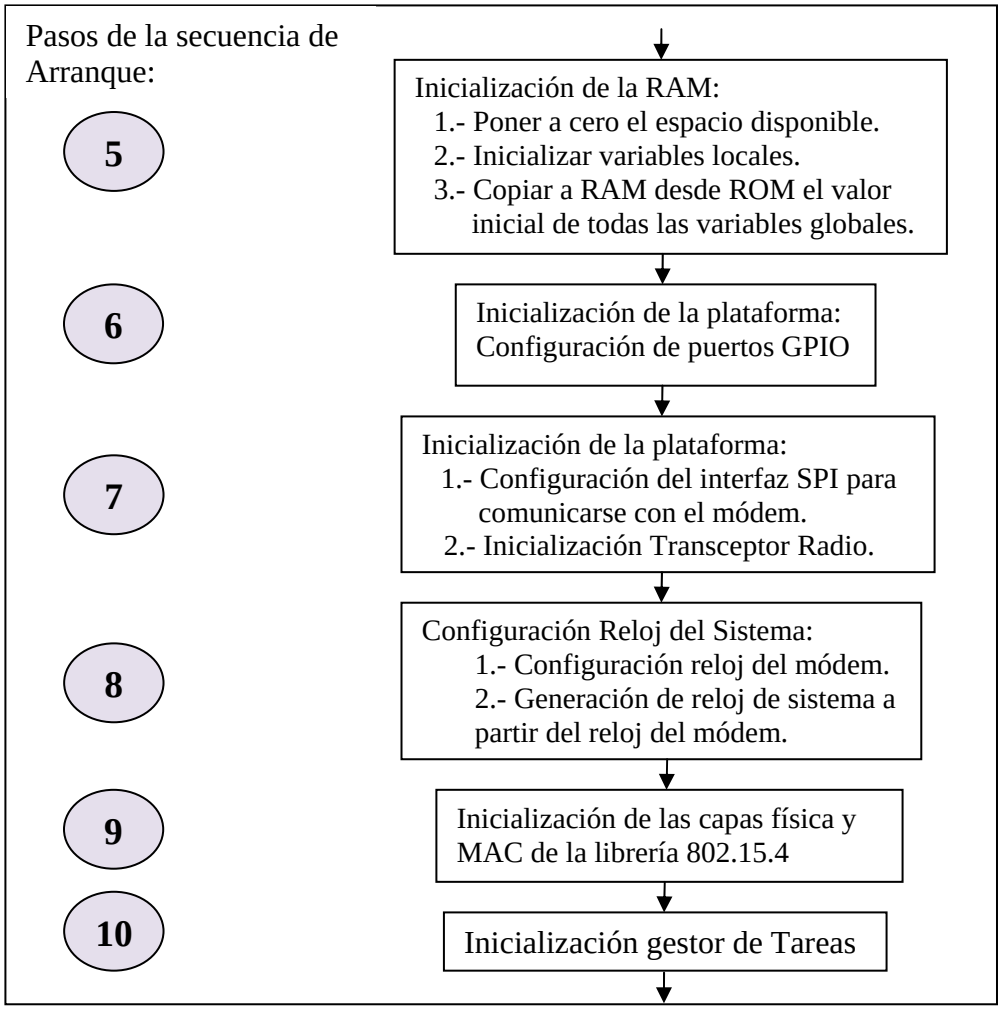
Principalmente cabe destacar los siguientes:

- La posición concreta del bootloader dentro de la secuencia de arranque determinará la frecuencia del reloj del bus interno que se empleará para la actualización del *firmware* (en el caso de requerirse una actualización), del que depende directamente el valor almacenado en el registro **FCDIV** para configurar correctamente la velocidad del reloj del módulo Flash entre 150 y 200 KHz. Este registro sólo puede escribirse en una ocasión por cada reseteo del dispositivo.
- Cuanto más próximo al inicio del proceso de arranque del dispositivo se posicione el bootloader, el reloj empleado será de menor frecuencia y menor precisión (al utilizar una fuente interna del MC13213). Esto podría tener una repercusión negativa en los procedimientos de escritura de la Flash y la EEPROM.

Si se analizan los pasos de la secuencia de arranque de la figura del apartado 4.1.2, se comprueba que cualquier localización del bootloader dentro de ésta podría ser válido una vez efectuado el paso 5, momento en que ya se ha inicializado la memoria RAM y el reloj interno del microcontrolador.

Por otro lado, en el afán de hacer al bootloader lo más independiente posible del código de aplicación (de forma que no sea nunca interrumpido por la aplicación principal cuando se está ejecutando una actualización), tampoco conviene emplazarlo en una posición de la secuencia de arranque posterior al paso 9 (inclusive). Esto se debe a que la inicialización del gestor de tareas que tiene lugar en el paso 10 podría afectar a la continuidad de la ejecución de una actualización o a que una tarea no se ejecute debidamente por inanición (si el bootloader acapara los recursos demasiado tiempo).

Así pues, el rango de pasos que comprende la secuencia de arranque del dispositivo que pueden ser aptos para el emplazamiento del bootloader se muestran en el siguiente diagrama:



Localizaciones posibles del bootloader dentro de la secuencia de arranque

De este modo, a continuación se analizarán los pros y contras de cada una de las 5 localizaciones posibles para el bootloader en la secuencia de arranque del dispositivo:

1. Después del paso 5: tras la inicialización de la memoria RAM empleado la fuente de reloj interna del microcontrolador como referencia. La principal ventaja de esta localización es que es la más independiente del código de la aplicación final. El principal inconveniente es que el reloj empleado es más lento e impreciso.
2. Después del paso 6: tras la inicialización de los puertos GPIO. Puesto que dichos puertos no afectan al bootloader (a menos que sean inconvenientemente configurados y se activen interrupciones indebidas), esta localización no presenta ninguna ventaja a la anterior (y podría plantear nuevos inconvenientes).
3. Después del paso 7: Tras la configuración del interfaz SPI y la inicialización del transceptor radio. Este paso activará la comunicación del microcontrolador con el módem que permitirá realizar emisiones y recepciones radio y emplear adicionalmente su señal de salida como nueva referencia de reloj (más rápida y estable). Sin embargo, justo después de este paso aún no se ha configurado el nuevo reloj del bus mientras que las interrupciones SPI ya podrían afectar al funcionamiento del bootloader. De nuevo esta localización no plantea ninguna ventaja nueva (y sí posibles inconvenientes) a la primera localización planteada.
4. Después del paso 8: tras la configuración de un nuevo reloj de bus a partir de la señal de referencia del modem radio (más rápida y estable). Las ventajas de esta localización son claras: mayor velocidad del bootloader y mayor precisión. Los inconvenientes: dependencia del bootloader de la frecuencia de bus elegida por el programador de la aplicación principal (una vez programado el bootloader ésta ya nunca podría modificarse) y la coexistencia con un mayor número de interrupciones y módulos activos que podrían alterar su correcto funcionamiento.
5. Después del paso 9: tras la inicialización de las capas física (PHY) y de acceso al medio (MAC) de la librería 802.15.4 suministrada. Puesto que el bootloader es independiente de ellas, su inicialización no plantea ninguna ventaja respecto a la localización anterior mientras que podría plantear algún inconveniente no considerado.

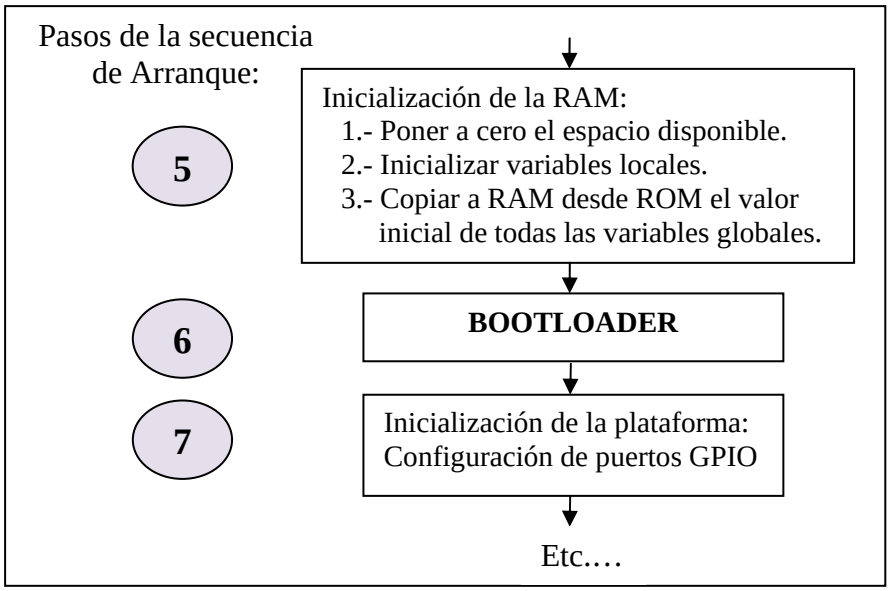
Así pues, de todos los posibles emplazamientos del bootloader dentro de la secuencia de arranque sólo dos parecen adecuados:

1. Justo después de la inicialización de la RAM (tras el paso 5 de la secuencia de arranque). Presenta la mayor independencia posible del bootloader con la aplicación final y las mínimas interferencias posibles de otros módulos del microcontrolador.
2. Justo después de la sincronización del microcontrolador con el módem radio y el establecimiento de la señal de reloj a partir de éste. Esta localización presenta la mayor velocidad de bus y el reloj más preciso que se emplearán en el microcontrolador.

Aunque ambas posibilidades son correctas, se ha decidido finalmente emplazar el bootloader dentro de la secuencia de arranque justo después de la inicialización de la memoria RAM (opción 1) por las siguientes razones:

1. El reloj interno empleado por defecto, a una frecuencia de 8 MHz de bus, es sólo la mitad de la que habitualmente se genera luego a partir de la señal de sincronización del módem radio (16 MHz).
2. Dicha diferencia de velocidades no afecta a la velocidad de borrado y grabación de memoria Flash que nunca será más rápida que 200 KHz. Esto es especialmente significativo puesto que el proceso de sobrescritura Flash es con diferencia el procedimiento más lento del bootloader. De esta forma, la velocidad del bus a efectos del bootloader sólo se apreciará a la hora de realizar lecturas y/o escrituras I^2C sobre la memoria auxiliar externa, la EEPROM, y no será significativa.
3. El empleo de una señal de reloj poco precisa no debería perturbar significativamente un protocolo de comunicaciones robusto como es el I^2C, diseñado para soportar características como el "estrechamiento de reloj". Simultáneamente, puesto que la señal de reloj del módulo Flash generada a partir del reloj de bus tiene una frecuencia 40 veces menor (200KHz máximo), la falta de precisión debería reducirse significativamente, dejando de ser un problema en la mayoría de los casos.
4. El bootloader ha sido programado para reintentar muchas veces cada operación que no se ejecute correctamente, de forma que está preparado para afrontar cualquier problema derivado de la falta de precisión.
5. Se ha comprobado el óptimo funcionamiento del bootloader en este emplazamiento y la sencillez de realizar dicha instalación dentro de la secuencia de arranque (pasos enumerados más adelante dentro de las especificaciones del bootloader, apéndice A.1.1).
6. El posicionamiento del bootloader cuanto más próximo al principio de la secuencia de arranque del dispositivo facilita su funcionamiento independiente con respecto a la aplicación principal, sus módulos en memoria y sus interrupciones asociadas. De este modo, cualquier programador podrá desarrollar fácilmente aplicaciones complejas ZigBee de forma transparente al bootloader.

Por todo esto se concluye como opción idónea el emplazamiento del bootloader dentro de la secuencia de arranque del ND07 justo después de la inicialización de la memoria RAM, como se muestra en el siguiente diagrama:



Emplazamiento definitivo del bootloader en la secuencia de arranque

Los pasos necesarios a dar por el desarrollador de aplicaciones ZigBee para emplazar el bootloader en la localización elegida dentro de la secuencia de arranque del ND07 son muy sencillos, y se enumeran en el apéndice A.1.1 de esta memoria.

4.1.5. Servicio Técnico de *Freescale*

El servicio técnico de *Freescale* consiste básicamente en un portal que mantiene un foro de dudas y un sistema de formularios donde los desarrolladores plantean incidencias directamente a la compañía.

Aunque a lo largo de los años este servicio ha mejorado bastante, durante el periodo de desarrollo de este proyecto su funcionamiento ha sido extremadamente desastroso.

Una consulta a *Freescale* tardaba de media dos semanas en ser atendida y cuando esto por fin sucedía por lo general la respuesta era de una simpleza elemental (básicamente una redirección a los manuales más básicos), dando clara muestra de no haber entendido la pregunta o estar muy lejos de saber la respuesta. Era necesario repetir la pregunta (dos semanas cada vez) hasta que la incidencia se escalaba por fin a un técnico especializado capaz de responder a la consulta con información de alguna utilidad.

De este modo, la mayoría de las consultas han llevado más de dos meses para obtener una respuesta más o menos orientada al tema de la pregunta y, desafortunadamente, muy a menudo ha sido para recibir la confirmación de un error de la librería ZigBee que *Freescale* prometía anotar para solventar en la próxima versión de ésta. Sin embargo, errores detectados como el mencionado sobre la NVM (ver apartado 3.8.3) nunca se solventaron en la pila ZigBee

2006 de *Freescale* (aunque llegó a reconocer este error al final en varias de las consultas que se le efectuaron).

Aunque la documentación que suministra *Freescale* para el chip MC13213 es considerablemente extensa, detalles concretos del funcionamiento del cargador de arranque, la memoria, las opciones de configuración del compilador, linkador y ensamblador del CodeWarrior son difíciles o imposibles de encontrar e interpretar salvo mediante consultas en los foros (si hay mucha suerte) o mediante el servicio técnico (extremadamente lento y en ocasiones inútil).

Cuando al principio del proyecto se programaron las primeras versiones funcionales del bootloader, todo apuntaba a que se lograría emplazar siempre todo su código en las páginas de memoria deseadas por el desarrollador software mediante directivas del ensamblador y opciones especiales del compilador.

Fue a lo largo del proceso de pruebas que se detectó que el código del bootloader sólo se conseguía independizar del resto de la aplicación en determinadas ocasiones (aleatorias). Al principio se perdió mucho tiempo creyendo que se estaban empleando mal las herramientas que ofrece *Freescale* para el desarrollo, compilación, linkado y ensamblado de programas.

No obstante, después de muchos intentos de obtener información al respecto, el servicio técnico confirmó que no se podía garantizar mediante opciones ni directivas del compilador que el código del bootloader se alojase en una sección concreta de la memoria Flash sin ninguna dependencia de otros sectores. Esto sucedía habitualmente porque estos sectores alojaban a su vez funciones de optimización de las operaciones realizadas en el proceso de actualización de *firmware* que el compilador asociaba al bootloader para reducir su tamaño y aumentar su velocidad.

Adicionalmente el servicio técnico confirmó que para lograr el objetivo buscado de independencia de código era absolutamente necesario implementar todo el bootloader directamente en ensamblador.

De este modo se puede concluir que el servicio técnico de *Freescale*, si bien ha servido para tomar decisiones de diseño fundamentales para el correcto funcionamiento del bootloader, ha requerido de la inversión de mucho esfuerzo y tiempo para logar este objetivo. Todos los problemas encontrados a lo largo del desarrollo podrían haber sido solventados o sorteados con un considerable menor coste en tiempo si el servicio técnico hubiese funcionado mejor y hubiese sido menos reticente a la hora de reconocer errores de sus librerías y de su documentación.

4.1.6. Actualizaciones de Librerías: Consecuencias y Estado Actual

Como consecuencia de los errores de la pila ZigBee 2006 reportados a *Freescale* a través de su sistema de incidencias y de las sucesivas mejoras que se han ido aplicando a la librería, la compañía ha publicado hasta cinco versiones de su *BeeStack* en menos de dos años.

Cada actualización ha implicado cambios tanto a las librerías 802.15.4 y ZigBee (cuyo código se presenta ya compilado en su mayor parte) como a las aplicaciones de ejemplo (que han de emplear diferentes interfaces, estructuras de datos y funciones para alcanzar los mismos objetivos).

Así pues, desde la primera versión de la pila ZigBee de *Freescale* para el MC13213, denominada *BeeStack 1.0.0*, hasta la última que se suministró a principios del 2008 (cuando *Freescale* dejó de dar soporte ZigBee 2006 para este chip), la versión 1.0.5, se han sucedido muchos cambios en el funcionamiento del API suministrado.

Muy a nuestro pesar, el empleo habitual del servicio de incidencias de *Freescale* nos ha permitido informar y corroborar un número considerable de errores (algunos muy críticos) en las librerías suministradas. De este modo, muchas veces ha sido necesario esperar a la publicación de una actualización de dichas librerías para tener un funcionamiento adecuado del sistema (con la consecuente pérdida de tiempo).

No obstante, cada publicación de una actualización de las librerías ha implicado un considerable trabajo adicional para comprobar que todas las funciones, estructuras y variables empleadas tanto por el bootloader como por las aplicaciones ZigBee seguían siendo compatibles. Por otro lado, no actualizar el proyecto nunca hubiese sido una opción, ya que cada versión de las librerías podía proveer de parches a fallos que todavía no habían sido detectados.

Es por ello que cada versión de las librerías 802.15.4 o ZigBee ha tenido que ser comprobada y el código de la aplicación ajustado para asegurar el funcionamiento de ésta idéntico (o mejor) al que ya se tenía. No bastaba con sobrescribir las librerías, el interfaz suministrado para ellas era habitualmente modificado por *Freescale* y no se documentaba adecuadamente.

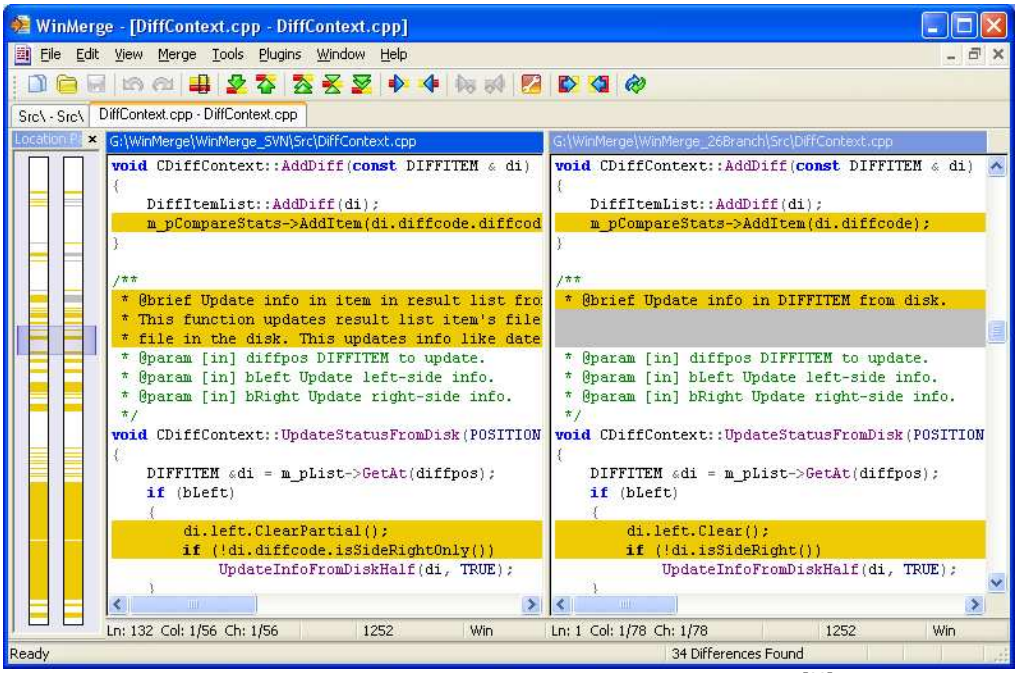
Obviamente, cada publicación de nuevas librerías implicaba a su vez la descarga de una nueva versión de la herramienta *BeeKit*, capaz de generar aplicaciones ZigBee de ejemplo con las nuevas pilas.

El procedimiento más seguro para actualizar correctamente las librerías ha consistido siempre en generar una nueva aplicación ZigBee de ejemplo con el BeeKit (con las nuevas librerías)

con los mismos parámetros originales que empleó la primera versión del bootloader y compararlo con el proyecto del bootloader en desarrollo.

Puesto que la aplicación ZigBee en proceso de desarrollo con el bootloader instalado siempre había sido modificada sustancialmente con respecto a la aplicación de ejemplo de la que partía (en parte debido a los numerosos fallos encontrados en los códigos y en las librerías), realizar dicha comparación a mano era considerablemente laborioso.

Así pues, ha sido necesario hacer uso de una herramienta que permitiese comparar ficheros organizados en árboles de directorios similares, resaltando las diferencias y facilitando su actualización. Para ello se ha empleado la aplicación libre *WinMerge*.



Captura de pantalla de la herramienta WinMerge [20]

Incluso mediante el empleo de la herramienta *WinMerge*, cada actualización de las librerías de un proyecto ha sido siempre un procedimiento delicado que ha requerido de la inversión de una considerable cantidad de tiempo y esfuerzo.

De este modo, a lo largo de la elaboración de este proyecto, ha sido necesario destinar varios días de trabajo a la actualización de las librerías y APIs cada vez que se recibía una publicación periódica.

A pesar de ello, *Freescale* nos ha anunciado su decisión de dejar de dar soporte a la librería ZigBee 2006 para esta plataforma y, desafortunadamente, la última versión publicada aún presenta fallos graves en el funcionamiento del protocolo ZigBee (que hemos detectado y la compañía ha reconocido).

4.2. Aplicación Auxiliar: Cargador RS232

4.2.1. Introducción y Objetivos

Como ya se ha mencionado anteriormente, el bootloader desarrollado en este proyecto se ha diseñado de forma que pueda funcionar de forma independiente al modo en que la aplicación principal (probablemente un sensor ZigBee) obtenga la imagen del nuevo *firmware*. Esto será verdad siempre y cuando dicha aplicación almacene la imagen en la EEPROM con el formato estipulado y cumpla todos los prerrequisitos especificados en este documento (ver apartado A.1.2).

Así pues, a la hora de implementar una aplicación capaz de recibir y almacenar imágenes, de las múltiples posibles, se optó por aquella que facilitase al máximo la depuración del bootloader (durante su fase de desarrollo) y que además requiriese de un esfuerzo moderado (ya que su fin principal sería la evaluación del proyecto).

De este modo, se decidió desarrollar una aplicación (de ahora en adelante la denominaremos "Cargador RS232") capaz de recibir y almacenar una imagen transmitida a través del puerto serie de un ordenador mediante el protocolo RS232 (ver apartado 3.10).

Esta elección se debió principalmente a las siguientes razones:

- El dispositivo de NLaza Soluciones ND07 permite una comunicación bidireccional completa conforme al protocolo RS232 de forma sencilla a través de su conector DE-9.
- *Freescale* suministra aplicaciones ZigBee de ejemplo con soporte de comunicaciones serie para el chip MC13213 integrado en el ND07. De este modo desarrollar una aplicación que actúe de pasarela ZigBee-RS232 es relativamente sencillo.
- Para que un dispositivo ZigBee reciba inalámbricamente una imagen de *firmware* tiene que existir previamente un dispositivo distribuidor de imágenes que las obtenga de forma cableada (mediante una comunicación serie, Ethernet, etc.). Así que el desarrollo de esta aplicación es imprescindible para toda demostración del bootloader.
- Todo el código de la aplicación sería reaprovechable en las ocasiones en que el dispositivo distribuidor de imágenes fuese un ND07 (en caso de querer desarrollar una aplicación comercial más robusta capaz de hacer transparente al usuario toda la comunicación).
- Dada la sencillez de la solución se podría depurar eficazmente el bootloader (objetivo último de este proyecto) con una considerable seguridad de que cualquier irregularidad encontrada durante el proceso vendría por parte del código del bootloader, maximizando la velocidad a la hora de encontrar y eliminar errores.

- Una vez finalizada la depuración del bootloader, esta sencilla aplicación servirá a la vez como demostración de su correcto funcionamiento.

Por tanto, el objetivo perseguido fue diseñar un cargador serie sencillo, intuitivo y de código reutilizable (aplicación escalable), que permita depurar el bootloader y finalmente demostrar su correcto funcionamiento.

4.2.2. Descripción de la Aplicación y Esquema de Funcionamiento

El cargador de imágenes RS232 es una aplicación ZigBee 2006 genérica de **interfaz combinado** (tipo de dispositivo "*Combined Interface*" (Id. 0x0007) según el perfil de automatización del hogar, "*Home Automation Profile*") que dispone adicionalmente de la capacidad de recibir y almacenar imágenes de *firmware* transmitidas desde el puerto serie de un ordenador.

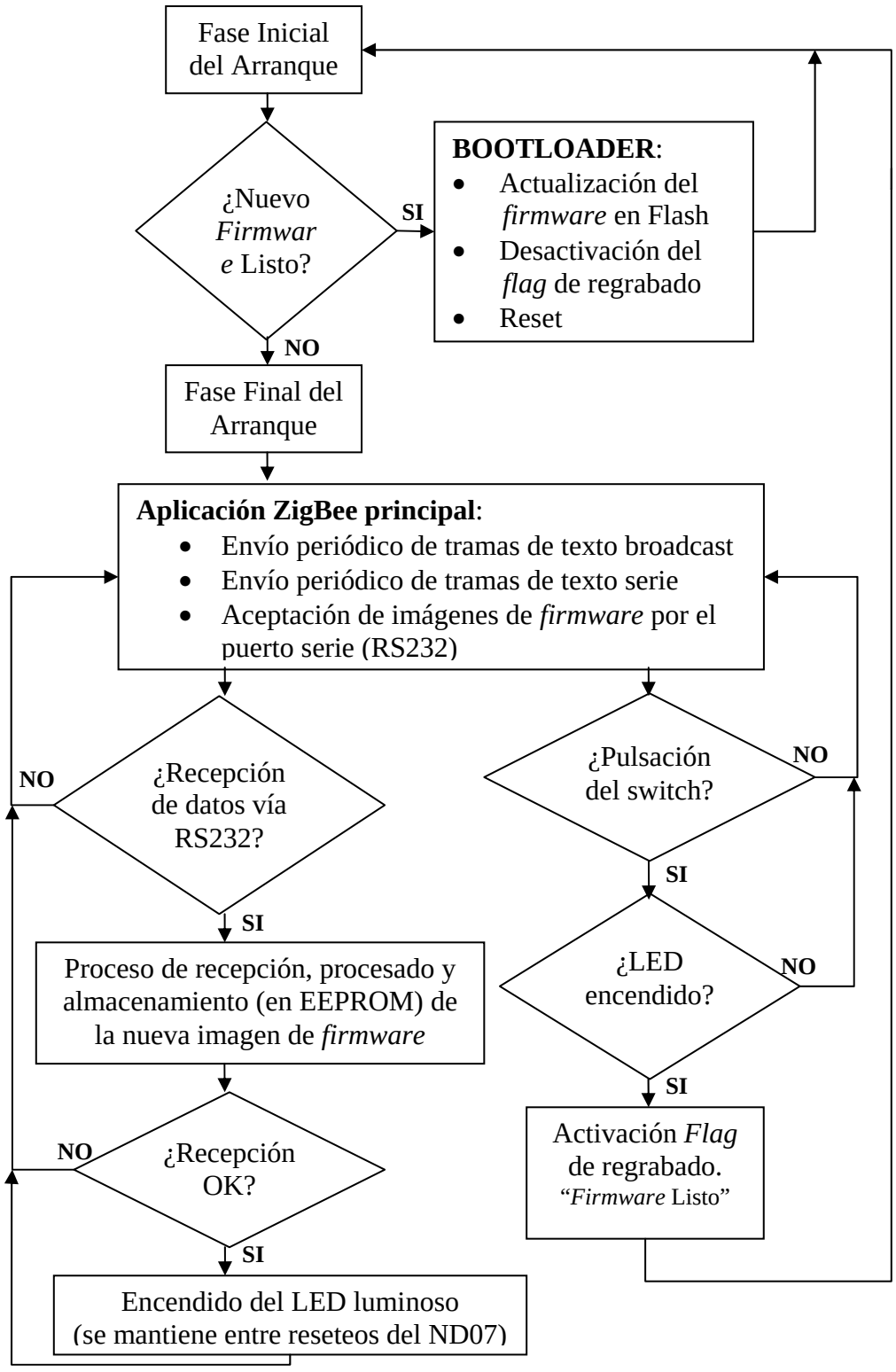
Además, el cargador de imágenes RS232 se ha programado como coordinador de red ZigBee (aunque esta característica podría modificarse en el futuro de forma muy sencilla) capaz pues de crear y gestionar una red mallada ZigBee. Esto se ha diseñado así para hacer al dispositivo independiente de otros dispositivos ZigBee (a excepción de otros coordinadores en el mismo canal) a la hora de crear una red y de enviar tramas broadcast sobre ésta (propiedad de la que se hará uso para validar el funcionamiento del bootloader)

De este modo, la aplicación desarrollada será capaz de:

- Arrancar y ejecutar (sólo ante un *flag* de regrabación activo) el bootloader.
- Crear una red ZigBee, aceptar solicitudes de asociación y proceder a enviar tramas *broadcast* sobre ella periódicamente.
- Enviar periódicamente tramas de información (o texto genérico) por el puerto serie para indicar el progreso de carga de una imagen (cuando proceda).
- Recibir una imagen de *firmware* transmitida a través de una conexión serie RS232.
- Procesar los registros S19 de la imagen, comprobar su integridad, traducirlos a registros S19 modificados y almacenarlos en la EEPROM auxiliar.
- Descartar los registros referidos a secciones de memoria no reescribibles (como son las páginas de la NVM o del propio bootloader) para evitar la necesidad de procesar previamente las imágenes antes de enviarlas.
- Indicar al exterior la existencia de una imagen válida en EEPROM que no ha sido empleada para actualizar el dispositivo mediante el encendido de un LED rojo.
- Permitir al usuario iniciar manualmente el proceso de actualización (si está el LED rojo encendido) mediante la pulsación de un interruptor (*switch*) del ND07. Esto provocará

la activación del *flag* de regrabación del dispositivo y posterior reseteo para que el bootloader realice la actualización del *firmware*.

Así pues, la máquina de estados del cargador RS232 podrá esquematizarse, para facilitar la comprensión de su funcionamiento, mediante la siguiente figura:



Máquina de estados del Cargador RS232

De este modo, se podría concluir que la aplicación de “Cargador RS232” se comportará como un coordinador de red ZigBee genérico (capaz de aceptar asociaciones de dispositivos a la red, enrutar paquetes, etcétera), con la peculiaridad de que emitirá periódicamente tramas de texto propietarias al canal inalámbrico (y al interfaz serie) y será capaz de recibir, procesar y almacenar una imagen de *firmware* para actualizar el dispositivo bajo demanda del usuario.

Esta aplicación, conceptualmente sencilla, nos permitirá demostrar el correcto funcionamiento del bootloader cuando se transmita una nueva versión del código a través de la comunicación RS232 y se solicite la actualización del *firmware* vía pulsación del interruptor del ND07.

Para simplificar el desarrollo de la aplicación de demostración, el cargador RS232, se ha diseñado de tal forma que se limita a recibir imágenes completas a través de la comunicación serie (sin tramas de negociación previas) y sólo da por bueno el nuevo *firmware* cuando ha recibido todos sus registros de forma íntegra (para ello comprueba el código de *checksum* de cada registro S19).

Para hacer lo más intuitiva posible la demostración, se transmitirá la imagen de una aplicación idéntica a la que tan solo se le haya modificado el texto de las tramas que envía a través del aire y de la comunicación serie. De este modo, la simulación será muy similar a una situación real en la que un programador decidiese actualizar la aplicación tras haberle realizado alguna pequeña mejora. Por otro lado, esta actualización podrá así comprobarse de forma inalámbrica mediante un Sniffer capturando tramas del canal elegido o de forma cableada mediante un terminal que permita la lectura y transmisión de tramas de texto vía RS232.

El API de comunicaciones I^2C con la EEPROM auxiliar programado durante el desarrollo del bootloader se reutilizó (en una versión ligeramente modificada y sin traducción a ensamblador ni código máquina) para lograr el almacenaje de las imágenes de *firmware* recibidas por el puerto serie del ND07.

La parte relativa a la comunicación RS232 de la aplicación se ha desarrollado basándose en el API de la UART que incluyen las aplicaciones ZigBee de ejemplo suministradas por *Freescale*. Puesto que el cargador RS232 se ha desarrollado a partir de un código de ejemplo de un interfaz combinado de automatización del hogar (*HA Combined Interface*), dicho módulo estaba incluido y sólo ha sido necesario aprender a utilizarlo.

Adicionalmente, ha sido necesario realizar todo el estudio previo sobre la tecnología ZigBee y la librería de *Freescale* descritos en los capítulos 2 y 3 para que, junto con la experiencia adquirida acerca del funcionamiento de sus aplicaciones de ejemplo, tener finalmente control sobre los distintos módulos que integra el cargador RS232:

- Control hardware del LED luminoso y puertos de entrada/salida en general (adecuación al caso del ND07).
- Puesta en marcha de una red ZigBee y el envío inalámbrico de tramas sobre ella.
- Temporizadores Software y Hardware (para el *watchdog* y el reset)
- Interrupciones de KBI (pulsadores) y rutinas de atención

Con todo esto, el cargador RS232 desarrollado permitirá simular una situación real de comportamiento de un coordinador ZigBee y validar así el funcionamiento del bootloader en un escenario de actualización de *firmware* a través de una comunicación serie RS232.

Adicionalmente, todo el código desarrollado podría ser reutilizado para desarrollar una aplicación distribuidora de actualizaciones de *firmware* a través de una red ZigBee.

Capítulo 5
Validación del Sistema

5.1. Introducción

El bootloader desarrollado en este proyecto ha requerido, como toda aplicación crítica que se programe, de un concienzudo proceso de depuración del código y ajuste de sus parámetros.

Aunque una de las propiedades más interesantes de la aplicación de bootloader es su independencia con respecto a la aplicación encargada de realizar la adquisición y almacenamiento de imágenes de *firmware*, es imprescindible programar ambas en el ND07 para poder depurar completamente su funcionamiento. De este modo, la depuración del bootloader se realizó siempre de forma conjunta con la depuración del cargador RS232.

De este modo, el proceso de depuración y validación del cargador RS232 con el bootloader instalado se realizó principalmente mediante 3 estrategias distintas (cada una de ellas especialmente adecuada para depurar un aspecto específico de la aplicación):

- Validación mediante el depurador del *CodeWarrior*
- Validación mediante trazas serie
- Validación mediante trazas radio ZigBee

A continuación se describirá cada uno de dichos procedimientos y finalmente se ilustrará el escenario de validación del sistema completo (demostración del proceso de actualización del cargador RS232 mediante el bootloader).

5.2. Validación Mediante el Depurador

El entorno de desarrollo del *CodeWarrior* incluye un depurador (*debugger*) bastante potente, sin embargo su empleo para la búsqueda y detección de errores del bootloader plantea los siguientes inconvenientes:

- Puesto que el depurador emplea su propio reloj de referencia para ejecutar el *firmware* del dispositivo y además permite realizar paradas (*breakpoints*) en la secuencia de ejecución, puede suceder que las temporizaciones requeridas para el borrado y regrabado de páginas de memoria Flash no se respeten adecuadamente y los datos se corrompan. De este modo, la depuración puede llegar a arrojar más confusión que pistas acerca de los errores del código (ya que introduce nuevos *bugs*).
- Lo mismo sucederá con la depuración de algoritmos de comunicaciones con periféricos que requieran de temporizaciones específicas (o *timeouts*). La interrupción (aunque sea momentánea) de la comunicación del MC13213 con otro integrado (mediante I^2C, RS232, etc.) puede provocar que el periférico descarte tramas, cierre comunicaciones

o simplemente se pierdan datos. Una vez más, la propia depuración puede llegar a introducir más incertidumbre que indicios acerca de los errores del código.

- El depurador funciona bajo la hipótesis de que el *firmware* del dispositivo es constante, de modo que emplearlo para validar una sustitución del *firmware* por parte del bootloader sólo puede realizarse de forma aproximada.

A pesar de todas estas limitaciones, el depurador del *CodeWarrior* sí que se ha empleado para depurar el funcionamiento del bootloader de forma global. Es decir, ha permitido comprobar que diversas secciones del código lograban sus objetivos parciales en determinados instantes concretos (por ejemplo el borrado completo de una página de la memoria, una vez éste ha finalizado) pero no ha facilitado la investigación profunda de todos los algoritmos involucrados por separado.

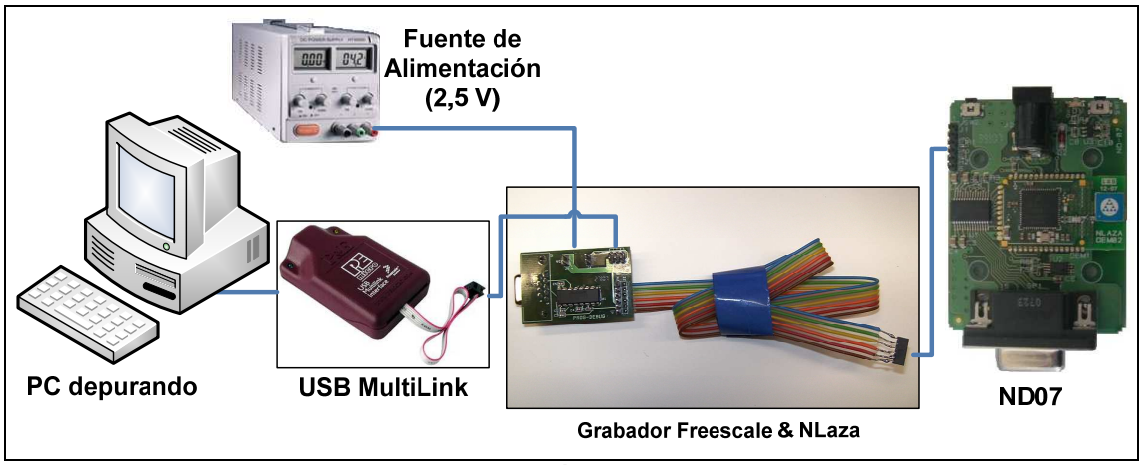
Sin embargo el depurador sí que ha permitido la depuración meticulosa de la aplicación principal ZigBee y de su algoritmo de recepción de imágenes (cargador RS232).

El proceso de validación mediante el depurador del *CodeWarrior* requiere de los siguientes elementos software y hardware:

- Ordenador (portátil o sobremesa) con el IDE *CodeWarrior for Microcontrollers v5.1* instalado y una licencia actualizada
- Dispositivo *USB MultiLink* de *Freescale* y sus *drivers* instalados en el PC.
- Dispositivo adaptador de NLaza Soluciones que permita emplear el *USB MultiLink* para programar el ND07 (adaptador NLaza/*Freescale*)
- Fuente de alimentación regulable capaz de suministrar 2,5 voltios de corriente continua

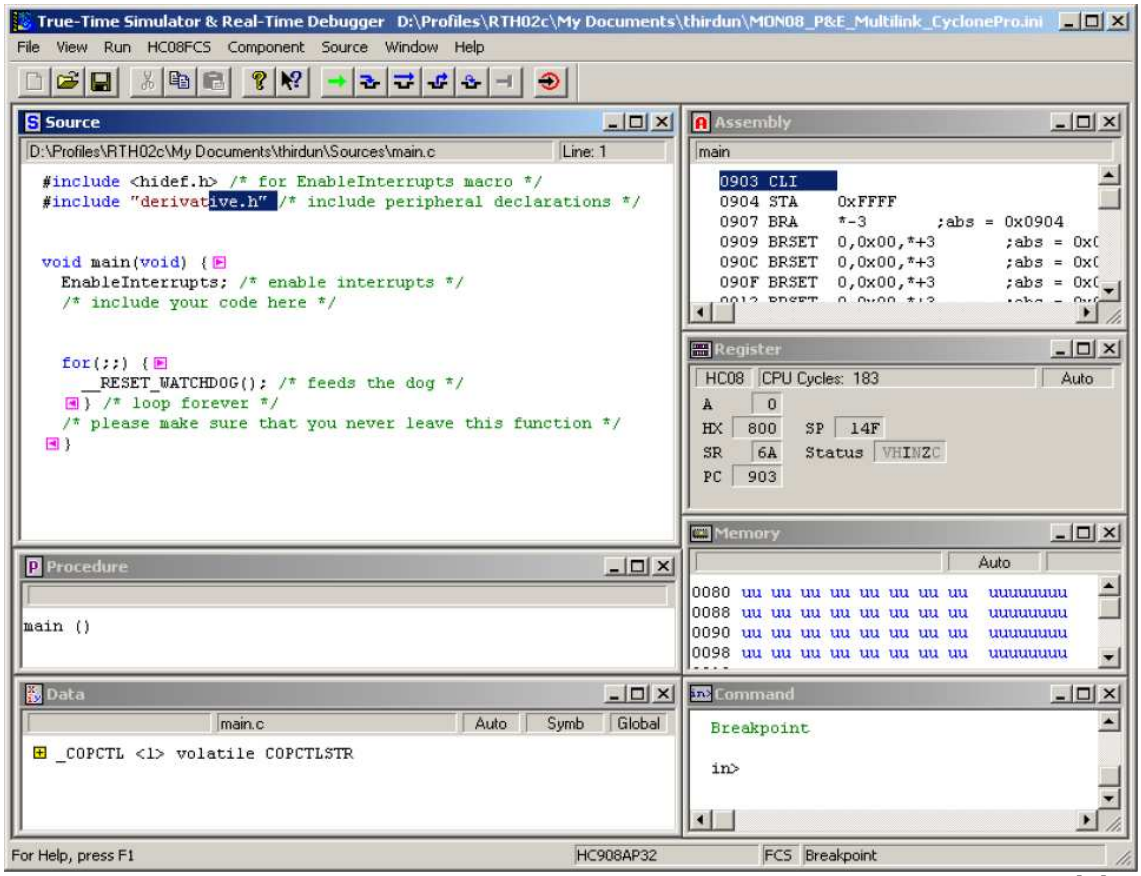
Aunque en situaciones de funcionamiento normal, para alimentar el dispositivo ND07 sólo se requiere de un adaptador de corriente a 4,5 voltios, mientras duren situaciones de programación del *firmware* y depuración de éste la alimentación ha de realizarse mediante una fuente de alimentación (regulada a 2,5 voltios) acoplada al adaptador NLaza/*Freescale*.

Aunque el apartado 3.5.3 ilustra profundamente dicho escenario de aplicación, éste puede esquematizarse mediante la siguiente figura:



Escenario de depuración del *firmware* del ND07

Una vez preparado este entorno de trabajo, el desarrollador compilará cada código a evaluar, lo programará en el ND07 e inicializará el depurador (*debugger*) incluido en el *CodeWarrior for Microcontrollers v5.1*, pudiendo visualizar así el siguiente conjunto de herramientas:



Captura de pantalla del funcionamiento del depurador incluido en el *CodeWarrior* [47]

Como puede apreciarse en la captura de pantalla, el depurador permite evaluar en un instante concreto de la ejecución de la aplicación (*breakpoint*) todas las variables de contexto relacionadas: mapa de memoria, registros del microcontrolador, valores de variables globales y locales, fuentes del código sin ensamblar y *firmware* ensamblado.

No obstante, como ya se ha mencionado, la interrupción de la ejecución secuencial del bootloader durante la actualización de un *firmware* habitualmente provoca que se corrompa el proceso (aunque arroje información puntual acerca del funcionamiento de una sección de código) y sea necesario después regrabar el dispositivo para seguir depurando.

Adicionalmente, el borrado y regrabación del *firmware* en tiempo de ejecución provocaba en muchas ocasiones que el depurador se bloquease (si se solicitaban *breakpoints* en determinados puntos del código) y era necesario reiniciar todo el proceso.

Este proceso de depuración ha sido pues un procedimiento muy lento y trabajoso que ha requerido en muchas ocasiones de la regrabación del ND07 y de la reinicialización del depurador decenas de veces para tratar de evaluar el funcionamiento de una sección concreta del código.

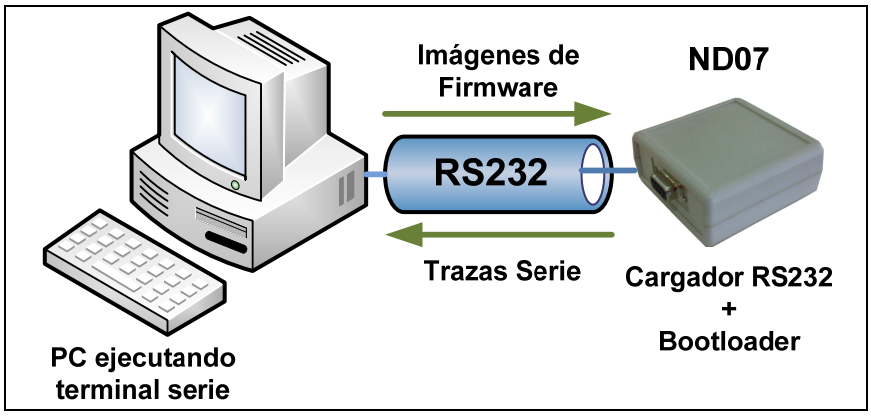
De este modo, la validación del bootloader (y del cargador RS2323) mediante el depurador se puede sintetizar como un lento procedimiento basado en modificaciones del código, evaluación de contextos de memoria y un gran número de regrabaciones del *firmware* y reinicializaciones de la aplicación del depurador.

5.3. Validación Mediante Trazas Serie RS232

El desarrollo simultáneo del bootloader en conjunto con la aplicación del cargador serie RS232 ha permitido, en un momento dado, depurar el funcionamiento de diversos módulos mediante la emisión de trazas a través de la comunicación RS232.

No obstante, la función más importante a desempeñar por la comunicación serie generada por el cargador RS232 ha sido la de recibir y almacenar imágenes de *firmware* en la memoria auxiliar. Gracias a esto, se ha podido depurar la sección del código del bootloader relativa a la lectura de registros modificados S19 desde la EEPROM para su posterior escritura sobre la memoria flash.

El escenario de validación mediante trazas RS232 ha requerido de un cable serie conectado desde la salida del ND07 hasta el puerto serie de un PC que se encontraba ejecutando una aplicación de terminal (*RealTerm*, *HyperTerminal*, etc.). El esquema se presenta en la figura siguiente:



Validación y depuración mediante trazas serie

Así pues, el proceso de validación mediante trazas serie simplemente ha consistido en la programación de la aplicación de forma que transmitiese el estado de determinadas variables de contexto (transmisión periódica o puntual) y se informase acerca del resultado de la ejecución de secciones concretas del código. De este modo, era posible estimar el correcto o incorrecto funcionamiento del bootloader y del cargador RS232 sin forzar una parada en su secuencia de ejecución (algo inevitable mediante el depurador) y obtener pistas acerca de los posibles errores.

Obviamente, una vez validado el correcto funcionamiento del bootloader y del cargador RS232 mediante las trazas serie, se han suprimido dichas trazas para la versión definitiva del código ya que, para el usuario final, no arrojarán información relevante (además se transmitirse ésta en un formato muy técnico y poco amigable).

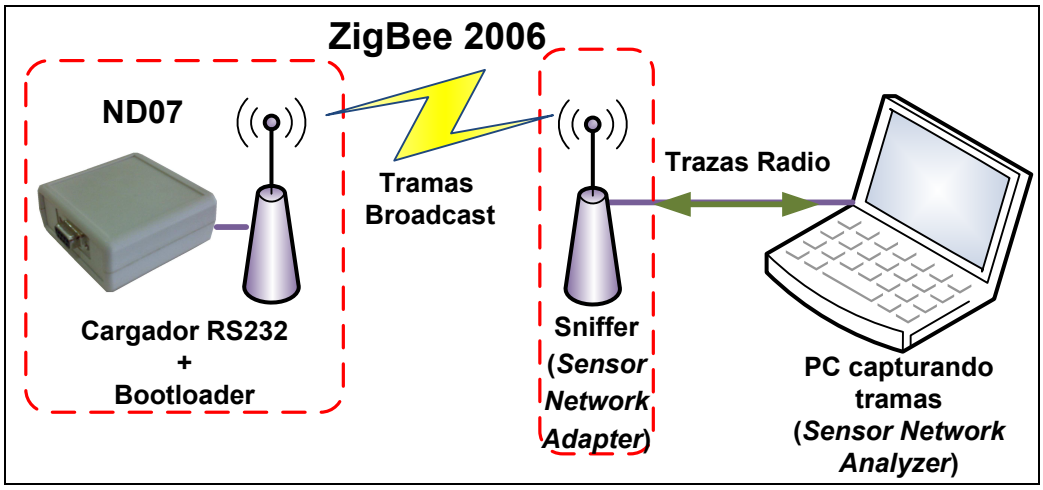
Adicionalmente, el cargador serie RS232 se ha diseñado se forma que emitirá trazas serie especiales a lo largo de todo el proceso de recepción de una imagen por el puerto serie, de forma que un usuario pueda saber en tiempo real el progreso del proceso y su estado. Estas trazas en cambio no se eliminarán ya que mejoran la experiencia del usuario del cargador RS232.

5.4. Validación Mediante Trazas Radio (ZigBee)

De forma adicional a los dos mecanismos de validación detallados en los subapartados anteriores, se ha validado el funcionamiento del código desarrollado (en este caso sólo del cargador RS232 ya que la radio no se inicializa hasta que el bootloader ha terminado de ejecutarse) mediante la emisión periódica (o eventual) de trazas radio *broadcast* con datos acerca del contexto de la aplicación. Una vez más, estos datos recibidos han mejorado la capacidad del desarrollador de detectar y corregir errores de la aplicación.

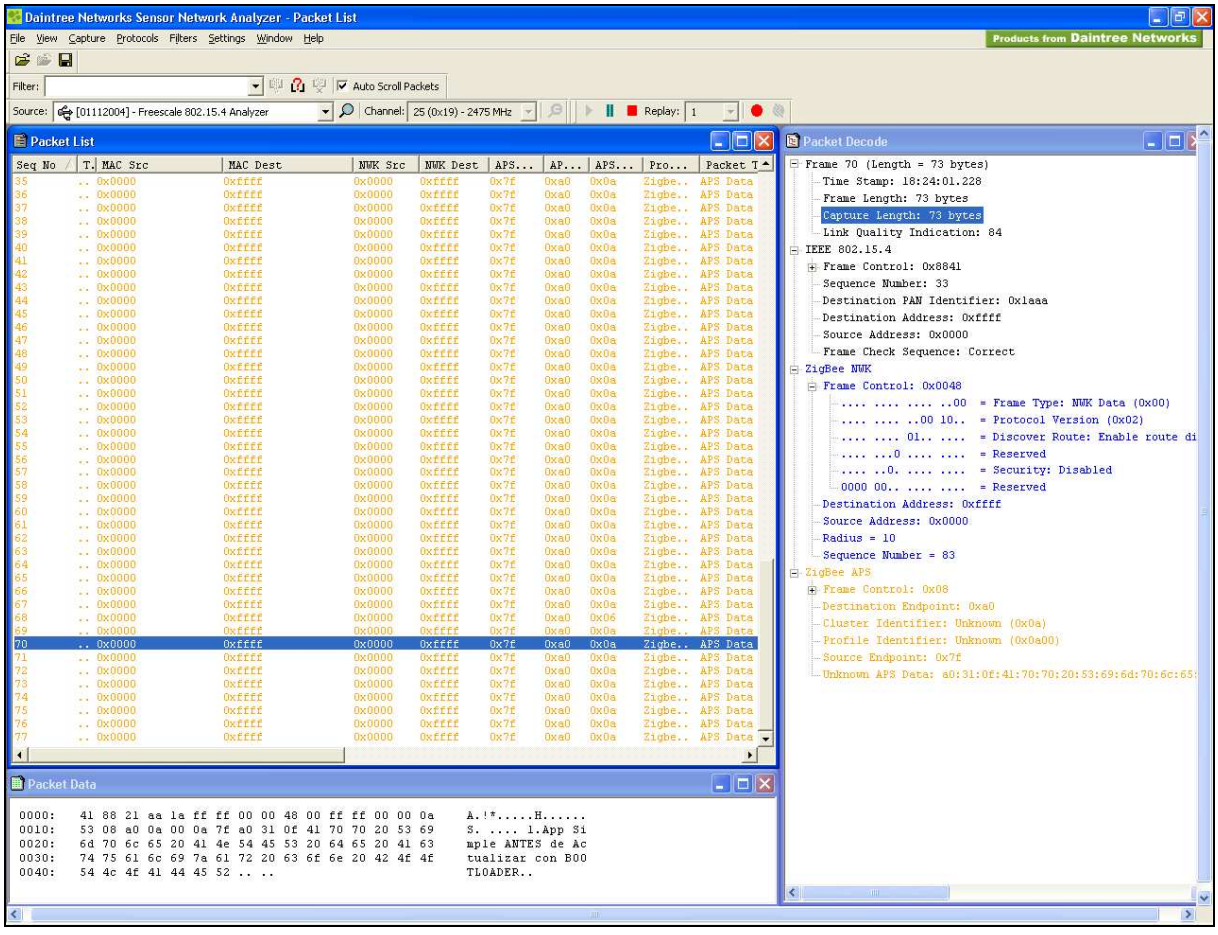
Este escenario de validación ha requerido de un ordenador (el mismo que realiza las trazas

serie u otro adicional) con la aplicación de captura de tramas SNA (*Daintree Sensor Network Analyzer*) ejecutándose y el sniffer (*Sensor Network Adapter*) correctamente acoplado e instalado. El escenario puede apreciarse pues en la figura siguiente:



Validación mediante captura de tramas radio

El software suministrado por *Daintree Networks* para la captura de tramas presenta un entorno de trabajo como el de la imagen siguiente:



Captura de pantalla del *Daintree Sensor Network Analyzer* (SNA)

De este modo, el proceso de validación ha consistido en el estudio de las tramas recibidas a través del sniffer radio, las cuales incluían información acerca de las variables de contexto de la aplicación bajo análisis.

De la misma forma que sucedió con las trazas serie, la trazas radio fueron suprimidas en la versión definitiva de la aplicación del cargador RS232, ya que su función había concluido satisfactoriamente.

Adicionalmente, el cargador serie RS232 se ha diseñado se forma que, mientras no se encuentre en proceso de actualización mediante el bootloader, emitirá tramas radio broadcast (no trazas) para indicar que ha formado una red de la que es coordinador ZigBee.

Estas tramas ZigBee incluirán un campo de texto legible mediante la aplicación de captura de paquetes (SNA) de modo que, si una actualización modificase su contenido (uno de los objetivos de la demostración del sistema global), este cambio podría apreciarse perfectamente.

5.5. Validación del Sistema Completo (Demo Cargador Serie RS232 + Bootloader)

Una vez se ha depurado y validado el adecuado comportamiento de los distintos bloques que conforman la aplicación del bootloader, es el momento de validar y demostrar su funcionamiento en conjunto con la aplicación encargada de recibir y almacenar actualizaciones de *firmware*, el cargador RS232.

El objetivo más ambicioso que se perseguirá en esta demostración será el de validar la actualización de una aplicación radio (empleando ZigBee 2006) realizada mediante el bootloader. La aplicación ZigBee 2006 a actualizar será, evidentemente, el propio cargador RS232.

Un objetivo secundario de esta demostración será el de validar la actualización de una aplicación serie, que también será el cargador RS232.

Obviamente, aunque ambos objetivos se alcanzarán mediante el mismo procedimiento, la validación/demostración de su correcto funcionamiento sólo será posible si se disponen de los medios materiales necesarios en el momento de la demostración.

De este modo, realizar la demostración de la actualización de una aplicación serie RS232, requerirá de los siguientes elementos hardware y software:

- PC con aplicación de terminal (*RealTerm*) para la transmisión de imágenes de *firmware*
- ND07 con el cargador RS232 y el bootloader instalados
- Cable serie conectado entre el ND07 y el puerto serie del PC

Por otro lado, realizar la demostración de la actualización de una aplicación radio ZigBee 2006, requerirá (además de todos los requerimientos anteriores) de los siguientes elementos hardware y software:

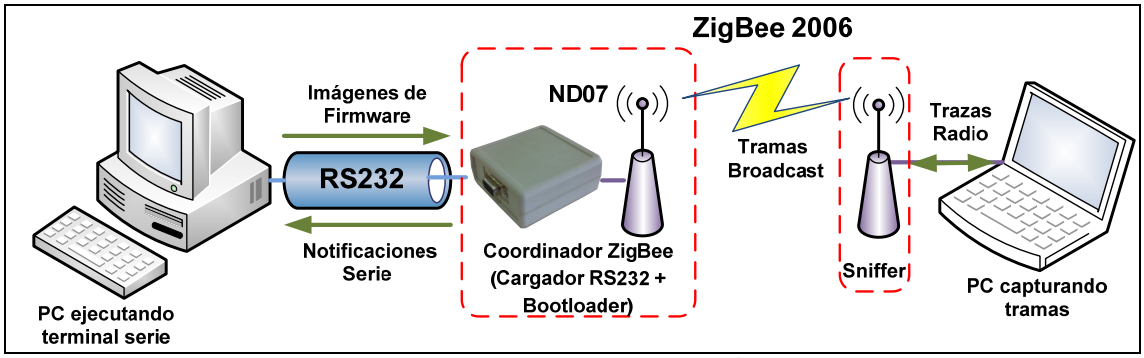
- PC con la aplicación de captura de tramas SNA (*Daintree Sensor Network Analyzer*) instalada y con licencia
- Sniffer (*Sensor Network Adapter*) correctamente acoplado e instalado

Puesto que el procedimiento de actualización del cargador RS232 es único, la demostración/validación sólo variará en complejidad y objetivos según el material disponible en el momento de la demostración.

Para que validación del sistema global sea realmente una demostración del funcionamiento del bootloader, se ha decidido que el nuevo *firmware* que sustituirá al original grabado en el ND07 sólo implementará modificaciones sobre los campos de texto emitidos periódicamente vía radio (ZigBee 2006) y vía comunicación serie (RS232). De este modo, al finalizar la actualización, un usuario podrá comprobar que el funcionamiento de la aplicación es idéntico salvo por la información contenida en dichos campos.

Este escenario será muy similar al de un sistema real, en el que las actualizaciones no suelen modificar significativamente el funcionamiento de la aplicación, sino sólo corregir errores e incluir ligeras mejoras, a veces inapreciables por el usuario.

De este modo, el escenario de la demostración/validación del cargador RS232 con el bootloader instalado puede esquematizarse mediante la siguiente figura:



Escenario de la demostración

Una demostración completa de todo el procedimiento de actualización requeriría de los siguientes pasos:

- Compilación y grabación del cargador RS232 (junto con el bootloader) sobre un dispositivo ND07 a través de las herramientas de grabación suministradas por *Freescale* (cable *MultiLink*).
- Demostración del funcionamiento de la aplicación programada (comunicaciones radio y por puerto serie).
- Realizar una modificación del código de la aplicación empleando el *CodeWarrior* que implique cambios apreciables desde el exterior por los usuarios.
- Recompilación y generación de una nueva imagen de *firmware* (compuesta por registros S19).
- Transmisión de dicha imagen a través del puerto serie del PC al cargador RS232 programado en el ND07.
- Comprobación del correcto funcionamiento de todo el proceso.
- Una vez finalizada exitosamente la transmisión serie, el cargador RS232 procederá a comprobar la integridad de la imagen recibida y que ha almacenado en la memoria auxiliar. En el caso de que este proceso se haya completado satisfactoriamente, el cargador encenderá un LED para indicar que está listo para actualizarse cuando el usuario así lo requiera.
- A continuación se pulsará el *switch* del dispositivo para solicitar la actualización del dispositivo. Esta pulsación forzará el apagado del LED.
- Se realizará después una espera activa hasta la finalización del proceso de actualización y el reseteo automático del dispositivo.
- Finalmente se comprobará de que la actualización se ha efectuado correctamente al aplicarse con éxito las modificaciones programadas (reflejadas en las comunicaciones radio y por puerto serie).

Desafortunadamente, seguir secuencialmente todos estos pasos suele requerir de una media de **entre 6 a 7 minutos**, y de la disponibilidad de varias aplicaciones y periféricos que no puede asegurarse.

Esto se debe a que tanto el *CodeWarrior* (empleado para la edición, compilación y grabación del código de ND07), como el software del *Sniffer* (para evaluar el funcionamiento de la comunicación ZigBee) requieren de licencias y periféricos específicos que podrían no estar disponibles en el momento de la demostración.

De darse esta situación, la demostración se podría simplificar haciéndola más sencilla y más corta en el tiempo. Para ello los nuevos pasos a dar serían los siguientes:

- Demostración del funcionamiento de la aplicación programada (sólo por puerto serie si no se dispone de *sniffer*). Esta aplicación (cargador RS232) se habría programado previamente en el ND07 en un laboratorio provisto de los medios necesarios.
- Transmisión de una nueva imagen de *firmware* a través del puerto serie del PC al cargador RS232 programado en el ND07. Este nuevo *firmware* sería una ligera variación del anterior y se presentaría compilado con antelación en un laboratorio.
- Comprobación del correcto funcionamiento de todo el proceso de transmisión de la nueva imagen vía RS232.
- En el caso de que este proceso se haya completado satisfactoriamente, el cargador encenderá un LED indicando que está listo para actualizarse cuando el usuario así lo requiera.
- A continuación se pulsará el switch del dispositivo para solicitar la actualización del dispositivo. Esta pulsación forzará el apagado del LED.
- Espera activa hasta la finalización del proceso de actualización que culminará con el reseteo automático del dispositivo.
- Comprobación de que la actualización se ha efectuado correctamente al aplicarse con éxito las modificaciones programadas (reflejadas en las trazas emitidas desde el cargador RS232 por el puerto serie y en las tramas ZigBee, en el caso de disponer de *sniffer*).

Esta modalidad de demostración, siendo considerablemente más teórica y menos visual, se completaría en un máximo de **cuatro minutos** teniendo en cuenta tan solo las temporizaciones de los procesos involucrados.

Una vez más, es necesario mencionar que la demostración del funcionamiento del bootloader estará supeditada a los medios materiales y a la cantidad de tiempo disponibles en el momento de su ejecución.

Capítulo 6
Conclusiones y Trabajos Futuros

6.1. Visión Global del Proyecto. Conclusiones

Como se introducía al principio de esta memoria, el objetivo principal de este proyecto ha sido el desarrollo de una aplicación de **bootloader** capaz de actualizar el *firmware* de dispositivos pertenecientes a una red de sensores 802.15.4 o ZigBee.

Este objetivo respondía a la clara necesidad de proveer de un mecanismo de actualización del *firmware* de los dispositivos ZigBee de la empresa *NLaza Soluciones,* que permitiese el empleo de cualquier protocolo de carga remota para hacer llegar las nuevas imágenes de *firmware* a los equipos.

Para alcanzar este objetivo, ha sido imprescindible realizar un profundo estudio previo de las tecnologías relacionadas, los entornos de desarrollo disponibles, las librerías suministradas, y las herramientas que proporcionan (para la reprogramación de la memoria) el dispositivo y el microcontrolador elegidos.

De este modo, el bootloader se ha programado en diversas fases (primero en C y luego se ha portado a ensamblador y código máquina) alcanzando todos los requerimientos impuestos por el dispositivo, las librerías y la tecnología: funcionamiento transparente con respecto a la aplicación principal del dispositivo, robustez, independencia del protocolo de telecarga, facilidad de instalación y bajos requisitos de memoria, de espacio disponible y de capacidad de proceso.

El correcto desarrollo del bootloader se ha logrado mediante la programación de una aplicación auxiliar, un cargador RS232 ZigBee, que ha permitido reproducir (por cable) el funcionamiento real de un protocolo de telecarga y así poder depurar y validar el proceso de actualización de *firmware*.

Así pues, fruto de este desarrollo se ha obtenido, además de las aplicaciones del bootloader y del cargador RS232, un manual para su instalación en dispositivos *NLaza Soluciones* funcionando con 802.15.4 o ZigBee 2006, y una serie especificaciones para el correcto almacenamiento de imágenes de *firmware* en la memoria EEPROM, que los diversos protocolos de telecarga programables deberán respetar para garantizar su compatibilidad.

A lo largo de todo el proceso de desarrollo se ha comprobado que las librerías 802.15.4 y ZigBee 2006 suministradas por *Freescale* para el microcontrolador MC13213, integrado en el dispositivo *NLaza ND07*, eran tal vez demasiado inmaduras.

Así pues, el proceso de desarrollo se ha visto ralentizado e interrumpido en innumerables ocasiones debido a la aparición de errores críticos en las librerías suministradas. El servicio técnico de *Freescale*, aunque ha ido mejorando con el paso de los años, ha demostrado su

ineficiencia y lentitud en la mayoría de los casos (el tiempo medio de respuesta a una consulta o notificación de error en la librería nunca fue inferior a una semana y, por lo general, requería de 4 o 5 respuestas por su parte para que *Freescale* reconociese una incidencia o plantease una solución de contingencia).

La numerosa publicación de revisiones y parches de la librería también ha sido un indicador de la inmadurez de la plataforma que, cuando surgió, se planteaba como la más prometedora de las disponibles en el mercado.

La observación de la evolución de la tecnología a lo largo de los años también sugiere que es ahora, con bastantes años de retraso, cuando por fin se empiezan a ver soluciones ZigBee disponibles para su adquisición. Las prestaciones de ZigBee PRO ofrecen soluciones inteligentes para muchos de los puntos débiles y limitaciones que planteaba ZigBee 2006, por lo que es de esperar que sea esta versión de la especificación la que se imponga en el mercado.

De igual forma, queda patente que el chip MC13213 de *Freescale* se queda corto, en cuanto a prestaciones se refiere, para la implementación adecuada del protocolo ZigBee PRO. Muestra de lo cual es el cese de soporte por parte de *Freescale* a las librerías ZigBee 2006 y la focalización de sus esfuerzos en las librerías PRO y en nuevas líneas de chips más potentes.

No obstante, 802.15.4 y ZigBee 2006 permiten el despliegue de redes de sensores inalámbricos de bajo consumo completamente funcionales y, el bootloader desarrollado, permitirá mejorar aún más sus prestaciones.

De este modo, el bootloader programado se ha empleado exitosamente como base para una aplicación de carga remota en redes 802.15.4, enmarcada dentro del proyecto de investigación **LoRIS** (localización en entornos socio-sanitarios) [48], financiado por el ministerio de Industria.

Adicionalmente, el bootloader se encuentra actualmente instalado en diversos modelos de dispositivos ZigBee de *NLaza Soluciones*, permitiendo la actualización de su *firmware* a través de diversos protocolos propietarios de telecarga.

Por otro lado, la aplicación del cargador serie RS232 será una base idónea para el desarrollo de un coordinador ZigBee encargado de la actualización remota de dispositivos empleando para ello las imágenes de *firmware* que recibe a través de la comunicación serie.

Como conclusión se puede afirmar que el proyecto del bootloader ha alcanzado sus objetivos y que su correcto funcionamiento ha sido convenientemente validado en escenarios de aplicación reales.

6.2. Trabajos Futuros

El bootloader y el cargador RS232 desarrollados en este proyecto sugieren una serie de futuras líneas de trabajo que se enumerarán a continuación.

1. **Creación de nuevos protocolos de carga remota** para distintos escenarios de redes de sensores inalámbricos basados en el bootloader desarrollado. Los protocolos más interesantes serán aquellos que se diseñen de forma que sean compatibles con los comandos y formatos de trama especificados por los perfiles públicos de aplicación (*Home Automation*, *Smart Energy*, etc.).
2. **Ampliación de las funcionalidades y capacidades del cargador RS232** para que sea capaz de gestionar la red ZigBee 2006 y funcionar como repositorio de actualizaciones para los dispositivos que la conforman.
3. **Desarrollo de nuevas aplicaciones** ZigBee 2006 que reproduzcan el funcionamiento del cargador RS232 pero **que permitan la adquisición de nuevas imágenes a través de otros interfaces** como RS485, USB o Ethernet, por ejemplo. Las imágenes recibidas se emplearían luego para distribuir inalámbricamente las actualizaciones entre los dispositivos de la red.
4. **Creación de una aplicación visual que permita la actualización remota** de dispositivos ZigBee 2006 **desde un centro de control** como puede ser un PC o similar. Esto implicaría el desarrollo de un protocolo de transmisión serie más robusto con reenvío de tramas y descarte de redundancias. El proyecto de fin de carrera desarrollado por R. G. Carranza para la carga remota en redes 802.15.4 (basado en este bootloader) puede servir de referencia [48].
5. **Modificación del componente NVM** de las librerías ZigBee 2006 para que realice el almacenamiento de datos de contexto en el espacio sobrante de la EEPROM auxiliar (un poco menos de 4K) y así ofrecer una solución de contingencia para los errores de las librerías para los que no se suministra parche.
6. **Migración del bootloader a las librerías de *Freescale* de ZigBee PRO**. De este modo, al menos los dispositivos finales (que requieren menos memoria y la nueva librería debería caber) basados en chips MC13213 serán compatibles y podrán asociarse a una red ZigBee PRO y disponer de medios para su actualización remota.

Apéndice A

Instrucciones y Recomendaciones de Instalación

A.1. Instalación del Bootloader

En este apéndice se detallará el conjunto de instrucciones necesario para lograr la instalación del bootloader dentro de una aplicación ZigBee y los requisitos de diseño de ésta última que han de cumplirse para garantizarla adecuada coexistencia entre ambos.

Aunque estas instrucciones y especificaciones han sido diseñadas para instalar el bootloader dentro de una **aplicación ZigBee 2006** (empleando la **pila *BeeStack 1.0.5***) programada sobre un **dispositivo ND07**, una leve modificación de éstas permitiría realizar dicha instalación sobre otros dispositivos y librerías a condición de que empleasen el mismo microcontrolador (el MC13123 de *Freescale*).

De este modo, el bootloader se ha instalado con éxito sobre otros equipos ZigBee de la línea NDimension de NLaza Soluciones (el ND01, ND02, ND04 y un largo etcétera) y sobre placas de desarrollo de *Freescale*.

Puesto que el bootloader es independiente del protocolo inalámbrico (o conducido) del que hace uso la aplicación principal, también puede instalarse sobre aplicaciones 802.15.4 o Simple MAC empleando las librerías correspondientes que proporciona *Freescale*. Esto también se ha comprobado satisfactoriamente.

A.1.1. Instrucciones de Instalación

A continuación se presentan las instrucciones a seguir para incluir exitosamente el bootloader dentro del proyecto de una aplicación 802.15.4 o ZigBee desarrollada mediante el IDE *CodeWarrior for Microcontrollers v5.1*:

1. Dentro del CodeWarrior, crear una nueva carpeta (opción "*Create Group…*") llamada "**bootloader**" dentro del proyecto (si es una aplicación ZigBee 2006, moverla después dentro del grupo "*BeeApps*" por coherencia).
2. Crear ese directorio físico (dentro del directorio "*BeeApps*") e incluir los ficheros:

   Bootloader.c

   Bootloader.h
3. Dentro del CodeWarrior, incluir dichos ficheros dentro de la carpeta **bootloader** y habilitar su compilación para todos los *targets* del proyecto para los que se desee instalar el bootloader.

   Para ello pulsar sobre el botón del CodeWarrior asociado al fichero **bootloader.c** siguiente:

Captura de pantalla del CodeWarrior for Microcontrollers v5.1

4. Apuntar el vector de interrupción I^2C a su rutina de atención dentro del bootloader. Para ello abrir el fichero **isrvectors.c** sito en el grupo: *PLM/Source/Common/Sys* (podría encontrarse en otro directorio en el caso de no ser un proyecto ZigBee 2006) y sustituir la línea:

```
Default_Dummy_ISR, /* vector 24 IIC control */
```

por las líneas siguientes:

```
#if Bootloader_Included
(void(*near)())0xF87D, /* vector 24 IIC control */
#else
Default_Dummy_ISR, /* vector 24 IIC control */
#endif
```

5. Incluir la llamada al bootloader dentro de la secuencia de arranque del dispositivo en la posición concreta acordada en el apartado 4.1.4.5.2. Para ello, abrir el fichero **crt0.h** sito en el grupo*: PLM/Source/Common/Interface* y sustituir las líneas:

```
#if (gBeeStackProject_c == 1)
#define CALL_MAIN_INTERFACE       Init();\
                                  RST_GetResetStatus(); \
                                  main(); /* Call user “main” function */
#else
#define CALL_MAIN_INTERFACE       Init();\
                                  main(); /* Call user “main” function */
#endif /*gBeeStackProject_c*/
```

por las siguientes:

```
#if (gBeeStackProject_c == 1) && (Bootloader_Included == 1)
#define CALL_MAIN_INTERFACE     Init();\
                                 RST_GetResetStatus(); \
                                 Nlaza_Bootloader(); \
                                 main(); /* Call user “main” function */
#elif (gBeeStackProject_c == 1)
#define CALL_MAIN_INTERFACE     Init();\
                                 RST_GetResetStatus(); \
                                 main(); /* Call user “main” function */
#elif (gBeeStackProject_c == 0) && (Bootloader_Included == 1)
#define CALL_MAIN_INTERFACE     Init();\
                                 Nlaza_Bootloader(); \
                                 main(); /* Call user “main” function */
#else
#define CALL_MAIN_INTERFACE     Init();\
                                 main(); /* Call user “main” function */
#endif /*gBeeStackProject_c*/
```

6. Incluir al principio de dicho fichero las líneas:

```
#if (Bootloader_Included==1)
#include “Bootloader.h”

#endif
```

7. Deshabilitar la protección y la securización de bloques para poder actualizar la memoria Flash. Para ello es necesario sustituir en el fichero **NVFlash.h** en el grupo *PLM/Source/NVM* las líneas:

```
#define NVFOPT_VALUE   0x02
#define NVFPROT_VALUE 0x98
```

Por las siguientes:

```
#define NVFOPT_VALUE    0x42

#define NVFPROT_VALUE 0xC0
```

8. Añadir para cada *target* del proyecto sobre el que se desee instalar el bootloader la siguiente opción de compilación (*Menú Edit-> Settings->Compiler for HC08*):

```
-Dbootloader_Included=1
```

De este modo, aquellos *targets* del proyecto que no deseen incluir el bootloader incluirán dicha opción de compilación con valor cero o no la incluirán (en cuyo caso el CodeWarrior notificará un *warning*).

9. **Configurar el fichero de parámetros** del enlazador (l*inker*) para que reserve para el bootloader las secciones de memoria RAM y Flash requeridas. Para ello es necesario modificar el fichero **BeeStack.prm** (o en el caso de no ser ZiBee 2006 el fichero de parámetros concreto que use el proyecto) sito en */PLM/PRM* siguiendo los pasos enumerados a continuación:

a) Reservar una sección (dentro de la etiqueta **SECTIONS**) de memoria RAM concreta (no compartida por otras aplicaciones) para el bootloader. Para ello es necesario incluir las siguientes líneas:

```
//RAM volátil reservada por el Bootloader
BOOTLOADER_RAM_SECTION          = READ_WRITE 0x0260 TO 0x02A4;
```

y modificar la sección de RAM restante consecuentemente a través de las líneas:

```
// All of RAM that isn’t reserved for something else.
APP_RAM                 = READ_WRITE 0x02A5 TO 0x1047;  // RAM
```

b) Reservar una sección de memoria Flash concreta (dentro de la etiqueta **SECTIONS**) para el bootloader. Para ello es necesario incluir las siguientes líneas:

```
//Vector de interrupción I2C para aplicación (siempre justo antes del bootloader)
I2C_VECTOR_SECTION         = READ_ONLY  0xF7FE TO 0xF7FF;

//Secciones BOOTLOADER (3 páginas)
NLAZA_BOOT                 = READ_ONLY  0xF800 TO 0xF85F;
```

```
NLAZA_CODE              = READ_ONLY  0xF860 TO 0xFDFF;
```

y modificar la sección de Flash restante consecuentemente a través de las líneas:

```
// Use this version if there are 3 NV storage pages.
APP_CODE_2              = READ_ONLY 0x2000 TO 0xF7FD;   // Flash

//Post Bootloader...
APP_CODE_3              = READ_ONLY 0xFE00 TO 0xFF4F;   // Flash
```

c) Asignar el código y las variables estáticas del bootloader, a sus secciones correspondientes. Para ello es necesario incluir las siguientes líneas dentro de la etiqueta **PLACEMENT**:

```
//BOOTLOADER
NLAZA_BOOTLOADER          INTO NLAZA_BOOT;
NLAZA_BOOTCODE            INTO NLAZA_CODE;
BOOTLOADER_RAM            INTO BOOTLOADER_RAM_SECTION;

//Puntero a rutina atención  interrupción I2C para la aplicación ppal
I2C_APP_VECTOR            INTO I2C_VECTOR_SECTION;
```

y, finalmente, modificar el emplazamiento de código genérico por defecto:

```
DEFAULT_ROM  INTO APP_CODE_0, APP_CODE_1, APP_CODE_2,APP_CODE_3;
```

Una vez finalizados los nueve pasos enumerados, el bootloader se considerará instalado y se arrancará en cada reseteo del dispositivo.

A.1.2. Requisitos de la aplicación ZigBee principal

Para que un programador de aplicaciones 802.15.4 o ZigBee sobre el ND07 (o sobre cualquier dispositivo de la línea *NDimension* de *NLaza Soluciones*) pueda instalar exitosamente el bootloader desarrollado a lo largo de este proyecto, su aplicación deberá cumplir una serie de requisitos de diseño previos.

Dichos requisitos, se enumerarán a continuación:

1.- La aplicación deberá disponer de suficiente memoria RAM y Flash libre como para alojar convenientemente el bootloader. El bootloader precisa de, al menos:

- 69 bytes de memoria RAM, ubicados forzosamente en el rango de direcciones **[0x0260 – 0x02A4]**.
- 3 páginas de memoria Flash (1536 bytes) obligatoriamente emplazados en el rango de direcciones **[0xF800 – 0xFDFF].**

2.- La aplicación que reciba y almacene las imágenes en la EEPROM deberá comprobar que los registros S19 que las conforman apuntan a direcciones de memoria en orden creciente. Ésta es la opción por defecto de un fichero de registros S19, pero podrían llegar desordenados al dispositivo a actualizar y la actualización no se realizaría correctamente.

3.- La aplicación deberá descartar los registros S19 de cabecera y fin (en el hipotético y poco práctico caso en que se hayan transmitido al dispositivo a actualizar). Sólo habrán de almacenarse en la EEPROM los registros S19 de datos (y convenientemente traducidos al formato de registros S19 modificados).

4.- La aplicación principal no deberá permitir que se inicie el proceso de actualización en los casos en que el nivel de batería del dispositivo sea bajo, ya que la interrupción de la alimentación durante un proceso de sustitución de *firmware* puede provocar que el equipo se dañe de forma irrecuperable.

5.- Debido a las limitaciones de tamaño de la memoria EEPROM discutidas en el apartado 4.1.3, la imagen de *firmware* no debe ocupar más de 63840 bytes una vez almacenada en EEPROM mediante registros S19 modificados. Así pues la aplicación receptora del nuevo *firmware* deberá comprobar este parámetro (si no lo comprueba previamente la aplicación que distribuirá dicha imagen y que también, presumiblemente, almacenará la imagen en EEPROM).

6.- Para garantizar una correcta "comunicación" entre el bootloader y la aplicación principal, la semántica de los valores almacenados en el *flag* de regrabación del la EEPROM ha de ser la siguiente:

- **Flag=0xAABB→** "La EEPROM contiene una imagen íntegra (con CRC correcto) que aun no ha sido volcada a Flash (pero que está programada para volcarse en el próximo reseteo del dispositivo)".
- **Flag=0xBBBB→**"La EEPROM contiene una imagen corrupta".
- **Flag=0xCCCC→**"La EEPROM contiene una imagen íntegra que ya ha sido empleada para actualizar el dispositivo".
- **Flag=0xDDDD→**"La EEPROM contiene una imagen íntegra que puede emplearse (o no) para actualizar el dispositivo cuando se desee. Sin embargo todavía no está programada para actualizar al dispositivo en el próximo reseteo.
- El resto de valores son ignorados por el bootloader pero pueden ser empleados por la aplicación principal para indicar otros estados distintos de la imagen almacenada.

7.- La aplicación programada no deberá activar el mecanismo de bloqueo de memoria ni de redirección de vectores o de lo contrario el bootloader dejará de funcionar correctamente. En cambio, la decisión tomada acerca de la securización (o no) de la memoria del dispositivo no afectará al funcionamiento de éste. Los pasos a dar para cumplir este requisito se detallan

en el punto 7 de las instrucciones de instalación.

8.- El programador deberá diseñar las aplicaciones destinadas a actualizar el *firmware* del ND07 de forma que el vector de interrupción I^2C apunte siempre a la misma posición de memoria del bootloader. Para alcanzar este objetivo es imprescindible efectuar el paso 4 de las instrucciones de instalación anteriores. Siempre que se esté ejecutando la aplicación principal (y no el bootloader), el vector de interrupción I^2C estará automáticamente redireccionado a las posiciones de memoria **[0xF7FE – 0xF7FF]** (sólo por el hecho de instalar el bootloader).

9.- Para garantizar el correcto funcionamiento de la instalación, la aplicación deberá activar el *watchdog* y configurarlo con la temporización más larga posible, 2^{18} ciclos de bus.

10.- En el caso de que la aplicación emplee el módulo NVM para almacenar variables no volátiles, éstas deberán ubicarse en las páginas 13, 14 y 15 de la memoria Flash. En el caso de necesitar páginas adicionales es necesario tener en cuenta que el bootloader podría borrarlas durante actualizaciones del *firmware*, de forma que los datos cuya conservación sea crítica deberán almacenarse únicamente en las páginas citadas.

Incluso en el caso de que la aplicación 802.15.4 o ZigBee desarrollada cumpla todos estos prerrequisitos de diseño, el desarrollador deberá seguir escrupulosamente todas las instrucciones de instalación del bootloader detalladas en el subapartado A.1.1.1 o de lo contrario no se podrá garantizar su correcto funcionamiento.

Como cabe esperar, la aplicación desarrollada para depurar y validar el bootloader, el cargador serie RS232, cumple de forma estricta todos los prerrequisitos de diseño e instalación enumerados en este apéndice.

A.2. Recomendaciones de Instalación

Una vez seguidas las instrucciones de instalación del bootloader junto con las especificaciones de diseño de la aplicación principal citadas anteriormente, se espera que el conjunto formado por el bootloader y la aplicación funcione correctamente.

No obstante, existen una serie de recomendaciones de diseño de la aplicación adicionales que, tomadas en consideración, pueden optimizar el funcionamiento del dispositivo, protegerlo ante situaciones excepcionales y alargar su vida útil.

De este modo, estas recomendaciones de diseño se enumerarán a continuación:

1.- Existe un número máximo de borrados y regrabados de la memoria ROM del dispositivo, de modo que conviene actualizarlo sólo cuando sea imprescindible (por no hablar de los riesgos que entraña toda actualización). Y lo mismo sucede con la EEPROM externa.

2.- Conviene que la aplicación encargada de activar el bootloader para la actualización del *firmware* de un dispositivo se cerciore previamente del nivel de batería de que dispone y no permita la actualización hasta que se pueda garantizar alimentación ininterrumpida durante todo el proceso. En el caso del ND07 esto no representará un problema significativo ya que está alimentado a la corriente alterna a través de un transformador. No obstante conviene tenerlo en cuenta y no actualizar el ND07 en entornos de alimentación fluctuante (como podrían ser días tormentosos con riesgo de corte de suministro eléctrico).

3.- No se recomienda activar la capacidad de "dormir" el procesador (propiedad ZigBee muy común que permite ahorrar batería) hasta la finalización de la secuencia de arranque del dispositivo (y con ella una posible actualización del *firmware* por parte del bootloader) para evitar todo riesgo posible de perturbar las temporizaciones internas requeridas.

4.- Toda aplicación desarrollada que lleve instalado el bootloader requerirá de la programación de un mecanismo capaz de alojar imágenes de *firmware* en la EEPROM auxiliar mediante el bus I^2C. Para que dicho mecanismo funcione correctamente el desarrollador deberá programar una rutina de atención a interrupción I^2C y apuntarla desde el vector I^2C redireccionado sito en las posiciones de memoria **[0xF7FE – 0xF7FF]** (ver configuración del fichero de parámetros .prm de este apéndice). Dadas las instrucciones de instalación del bootloader del anterior subapartado, la forma más sencilla de lograr dicho objetivo será mediante las siguientes líneas de código:

```
typedef void(*ISR_funct_t)(void);


#pragma CONST_SEG I2C_APP_VECTOR
const ISR_funct_t I2C_vector[]={
  App_IIC_Interrupt         //Cambiar por el nombre concreto de la función programada
};
#pragma CONST_SEG DEFAULT
```

5.- El ND07, como la mayor parte de dispositivos ZigBee, dispone de recursos de memoria y capacidad de procesamiento reducidos. Es por esto que todo sistema inteligente de actualización de dispositivos en un red deberá ser diseñada de forma que el esfuerzo de procesamiento requerido por los dispositivos receptores de imágenes sea mínimo. La aplicación encargada de transmitir imágenes de *firmware* debería evitar el envío de datos superfluos en la medida de lo posible y el formato elegido para la transmisión de imágenes no debería requerir de una transformación muy compleja en recepción para poder almacenarlas.

El seguimiento de estas recomendaciones, junto con los requisitos e instrucciones de instalación de los subapartados anteriores, garantizará el desarrollo de un sistema de actualización de dispositivos eficiente, robusto y duradero.

Apéndice B

Presupuesto

B.1. Fases del Proyecto

Para poder evaluar el coste de personal invertido en la realización de este proyecto, es necesario separar en la medida de lo posible las distintas fases de ha comprendido, ofreciendo así una estimación temporal al mismo.

Esta separación será meramente orientativa ya que, como en todo proyecto de ingeniería, ciertas tareas se han solapado y ha existido una constante realimentación de las conclusiones de cada tarea para la redefinición y optimización de diseños planteados en tareas previas.

De este modo, las tareas realizadas en este proyecto se pueden desglosar, cronológicamente, de la siguiente forma:

1. **Estudio del estado del arte de las tecnologías de redes de sensores de bajo consumo**. Estudio en detalle del estándar IEEE 802.15.4 y la especificación ZigBee. Investigación acerca de las plataformas de desarrollo disponibles para 802.15.4 y ZigBee en el mercado (principales fabricantes). Selección del chip MC13213 de *Freescale* y del ND07 de *NLaza Soluciones* como plataformas de trabajo.
2. **Familiarización con el entorno de desarrollo**. Evaluación del IDE de programación (*CodeWarrior*) y de las librerías de 802.15.4 y ZigBee suministradas por *Freescale* para el chip MC13213. Generación de aplicaciones de ejemplo (herramienta *BeeKit*) sobre el kit de desarrollo y entrenamiento para la programación y depuración de placas mediante los periféricos incluidos en el kit. Familiarización con el API de las diferentes capas y el "sistema operativo", formado por el programador de tareas, el sistema de eventos y las interrupciones hardware. Evaluación del funcionamiento del bajo consumo y de la secuencia de arranque de las aplicaciones.
3. **Estudio del chip MC13213 de *Freescale***. Estudio de los diferentes módulos y controladores que integra. Aprendizaje de su manejo y configuración. Evaluación de la relación de cada uno de ellos con el bootloader y el RS232. Estudio concienzudo del funcionamiento de la memoria y de las herramientas disponibles para borrarla, regrabarla, protegerla o securizarla mediante el API incluido en las librerías 802.15.4 y ZigBee.
4. **Estudio del coordinador ZigBee ND07 de *NLaza***. Estudio de los componentes que integra (entre los que se encuentra el propio chip MC13213, un chip RS232, una EEPROM auxiliar y varios periféricos como LEDs o pulsadores) y su funcionamiento. Evaluación de las herramientas disponibles en las librerías para su control y gestión desde el microcontrolador (módulos de comunicación I^2C y UART principalmente). Estudio de los protocolos serie RS232 e I^2C para el manejo de la comunicación serie y la EEPROM auxiliar respectivamente. Evaluación y diseño del formato óptimo de almacenamiento de imágenes *firmware* en la EEPROM auxiliar.

5. **Programación del Bootloader en C**. Diseño de los diferentes bloques que requiere, programación de un API para las comunicaciones serie RS232 e I^2C (para la memoria auxiliar externa al chip). Programación en C de los bloques principales del bootloader, capaces de la lectura de *firmware* almacenado en la EEPROM y de la regrabación de la memoria Flash en tiempo de ejecución. Estudio del emplazamiento óptimo del bootloader dentro de la secuencia de arranque del dispositivo y la configuración de sus parámetros necesaria asociada.
6. **Localización, reproducción y resolución de errores de las librerías de *Freescale***. Múltiples conversaciones con el servicio técnico de *Freescale* vía correo electrónico durante meses para conseguir el reconocimiento de errores críticos de sus librerías y así obtener parches o soluciones de contingencia. Publicación de actualizaciones periódicas de las librerías por parte de *Freescale* que requerían un considerable esfuerzo para el portado del proyecto bajo desarrollo.
7. **Programación del cargador RS232 (aplicación auxiliar anfitriona del bootloader).** Diseño y programación de la aplicación, basada principalmente en el bloque de comunicaciones ZigBee y en el módulo de la UART para la comunicación RS232. Instalación del bootloader dentro de su secuencia de arranque.
8. **Portado del Bootloader a ensamblador y código máquina.** Única solución posible para asegurar el funcionamiento del bootloader de forma independiente a la aplicación anfitriona sobre la que se instale.
9. **Validación y depuración del cargador RS232 y el bootloader.** Detección y corrección de errores mediante las herramientas suministradas por *Freescale* y mediante la emisión de trazas radio y serie. Distintos escenarios de pruebas.
10. **Documentación de proyecto**. Generación de la documentación asociada al trabajo desarrollado. Generación de un manual de instalación (apéndice A). Generación de transparencias para la presentación del proyecto.

Como puede observarse, aproximadamente la mitad de las tareas enumeradas tienen un considerable estudio teórico asociado, lo cual se debe a que gran parte del esfuerzo realizado ha requerido de una profunda investigación acerca del funcionamiento de las diferentes tecnologías y módulos involucrados (según se iba haciendo necesaria).

Todas las tareas enumeradas han sido desarrolladas extensamente en el presente documento, pero siguiendo una ordenación ligeramente diferente que se describe en el capítulo 1.3.

La planificación de la ejecución de las tareas de este proyecto (y el solapamiento temporal que tiene lugar entre algunas de ellas), se ilustra en el siguiente diagrama de Gantt:

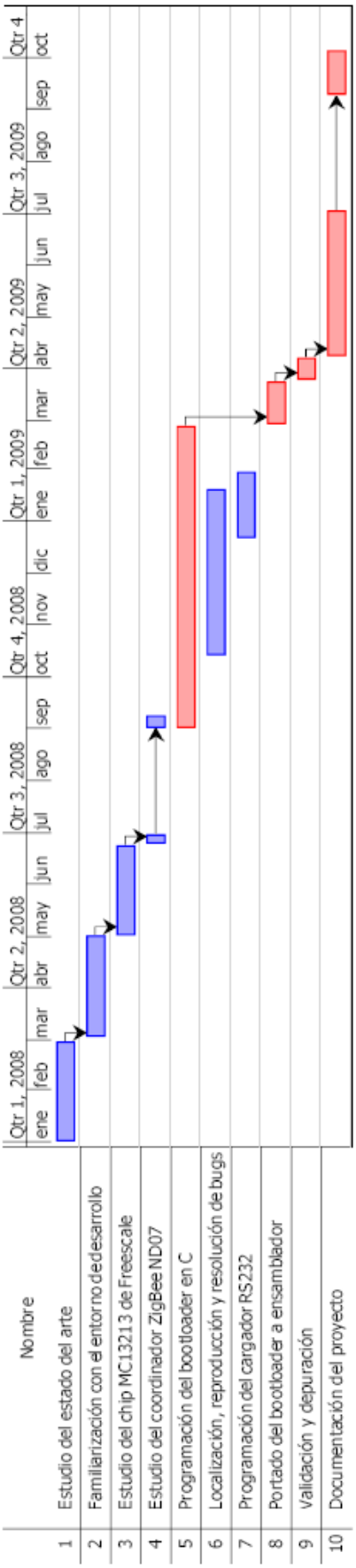


Diagrama de Gantt: Planificación del Proyecto

B.2. Costes de Recursos Humanos

Este apartado recogerá los costes asociados al trabajo de las personas implicadas en el proyecto. Para este cálculo, se considerará que todo el trabajo ha sido realizado por un ingeniero superior de telecomunicación.

El salario por hora de un ingeniero superior de telecomunicación estimado será de 40 euros/hora, lo que incluirá el sueldo que el ingeniero percibe, I.R.P.F, seguridad social y otros gastos de los que la empresa debe hacerse cargo.

A continuación la siguiente tabla presentará, basándose en la enumeración de las fases del proyecto descritas en el apartado anterior, la estimación del tiempo que ha requerido cada tarea y su coste asociado:

Tarea	Horas	Coste/Hora	Coste Total
Estudio del estado del arte	176 h	40 €/h	7040 €
Familiarización con el entorno de desarrollo	176 h	40 €/h	7040 €
Estudio del chip MC13213 de *Freescale*	150 h	40 €/h	6000 €
Estudio del coordinador ZigBee ND07	40 h	40 €/h	1600 €
Programación del bootloader en C	264 h	40 €/h	10560 €
Localización, reproducción y resolución de *bugs* en las librerías de *Freescale*	176 h	40 €/h	7040 €
Programación del cargador RS232	40 h	40 €/h	1600 €
Portado del bootloader a ensamblador y código máquina	72 h	40 €/h	2880 €
Validación y depuración	40 h	40 €/h	1600 €
Documentación del proyecto	328 h	40 €/h	13120 €
TOTAL	**1462 h**		**58480 €**

Costes de los recursos humanos

Para que esta estimación sea coherente con el diagrama de Gantt del apartado anterior, es importante señalar que la jornada de trabajo habitual ha sido de 4 horas los días laborables de la semana. Esto se debe a que el proyecto se ha realizado compaginándose con una jornada laboral completa en la compañía *NLaza Soluciones*.

B.3. Costes de Equipo Hardware

Basándose en el listado del capítulo 1.4, que enumera los dispositivos y equipo hardware que han sido necesarios para el desarrollo de este proyecto, se puede estimar el coste asociado al material en la siguiente tabla:

Equipo Hardware	Unidades	Precio Unitario	Precio Total
Ordenador de sobremesa	1	479.00 €*	479.00 €
Kit de desarrollo MC1321x de *Freescale*	1	1267.98 €	1267.98 €
***NLaza NDimension* ND07**	1	60.00 €	60.00 €
Programador/Depurador NLaza-*Freescale*	1	30.00 €	30.00 €
Sniffer Daintree Network Adapter	1	0.00 €**	0.00 €
Fuente de alimentación regulable	1	125.00 €	125.00 €
Cable de puerto serie	1	2.50 €	2.50 €
TOTAL (sin IVA)			**1964.48 €**
Base imponible (IVA 16%)			314.32 €
TOTAL (con IVA)			**2278.80 €**

Costes del equipo hardware

*Coste actualizado a octubre 2009 para un PC de sobremesa *Intel Pentium Dual Core* E5300 (2.6GHz, 800Mhz, 2MB) de la marca *Dell*.

**El coste del *sniffer* hardware se incluye dentro del kit de captura de tramas software más adelante.

B.4. Costes de Equipo Software

Basándose en el listado del capítulo 1.4, que enumera las aplicaciones software y las licencias que han sido necesarias para el desarrollo de este proyecto, se puede estimar el coste asociado al equipo software en la siguiente tabla:

Equipo Software	Unidades	Precio Unitario	Precio Total
Microsoft Windows XP Professional Ed.	1	120.65 €	120.65 €

IDE *CodeWarrior for Microcontrollers* v5.1 con una licencia estándar	1	666.01 €	666.01 €
Servidor de licencias flotantes *FlexLm* v8.4	1	0.00 €*	0.00 €
Analizador de protocolos *Daintree Sensor Network Analyzer* (SNA) con una licencia estándar	1	1335.36 €	1335.36 €
***BeeKit Wireless Connectivity Toolkit* con una licencia estándar**	1	866.81 €	866.81 €
Pila *BeeStack 1.0.5*.	1	0.00 €**	0.00 €*
***WinMerge* 2.0.2**	1	0.00 €	0.00 €
***RealTerm* 2.0.0.57**	1	0.00 €	0.00 €
TOTAL (sin IVA)			**2988.83 €**
Base Imponible (IVA 16%)			478.21 €
TOTAL (con IVA)			**3467.04 €**

Costes del equipo software

* El coste del servidor de licencias se incluye dentro del coste del IDE *CodeWarrior* con una licencia flotante estándar.

** El coste de la pila ZigBee y 802.15.4 están indirectamente incluidos en el coste de la licencia del *BeeKit*.

B.5. Costes Totales

Finalmente, basándose en los costes calculados de equipo (hardware y software) y de recursos humanos, se concluye el siguiente presupuesto del proyecto:

Concepto	Coste (€)
Recursos Humanos	58480.00 €
Equipo Hardware	1964.48 €
Equipo Software	2988.83 €
TOTAL (sin IVA)	**63433.30 €**
Base imponible (IVA 16% del equipo)	792.53 €
TOTAL (con IVA)	**64225.80 €**

Presupuesto del proyecto

Apéndice C
Glosario y Acrónimos

ACK	Asentimiento (*Acknowledgement*).
Ad Hoc	Tipo de red de control descentralizado en la que cada dispositivo participa en el encaminamiento de paquetes de otros nodos.
ADC	Conversor analógico-digital (*Analog-to-Digital Converter*).
AF	Plataforma de aplicación. Nivel de la arquitectura ZigBee encargado de alojar los diferentes objetos de aplicación y facilitarles diferentes interfaces (*Application Framework*).
AIB	Base de datos de nivel de aplicación (*Application Information Base*).
ALOHA	Protocolo de comunicaciones de nivel 2 en el modelo OSI. Describe un mecanismo de acceso al medio basado en el envío inmediato de datos y luego una escucha del canal en busca de colisiones (en cuyo caso se reintentará más tarde).
AP	Punto de acceso (*Access Point*).
APDU	Unidad de datos del protocolo de aplicación (*Application Protocol Data Unit*).
API	Interfaz de programación de aplicaciones (*Application Programming Interface*).
APS	Subnivel de soporte a la aplicación (*Application Support Sublayer*).
ASCII	*American Standard Code for Information Interchange.*
ASK	Técnica de modulación en la que se representan los datos como variaciones de la amplitud de una portadora (*Amplitude-shift Keying*).
Beacon	Trama baliza. Empleadas habitualmente para sincronizar dispositivos o para publicar las características de una PAN.
BeeStack	Nombre que recibe la implementación de la pila del protocolo ZigBee por parte de las librerías de *Freescale.*
Binding	Ligaduras. Asociaciones lógicas establecidas entre dos endpoints de una red ZigBee.
Bluetooth	Especificación industrial IEEE 802.15.1, que define un estándar global de comunicación inalámbrica de voz y datos entre diferentes dispositivos.
Breakpoint	Punto de ruptura. Secciones del código donde se interrumpirá temporalmente la secuencia de ejecución de instrucciones para depurar la aplicación.
Broadcast	Envío de información que será recibida por todos los dispositivos de una red.
BSN	Número de secuencia de baliza (*Beacon Sequence Number*).
Bug	Error de una aplicación o librería.
CCA	Estimación de canal libre (*Clear Channel Assessment*).
Chirp	Señales que disminuyen o aumentan de frecuencia con el tiempo. Se emplean habitualmente en técnicas de espectro ensanchado CSS.
CRC	Comprobación de redundancia cíclica (C*yclic Redundancy Check*).
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidance.*
CSS	Técnica de espectro ensanchado que emplea pulsos *chirp* para codificar la información *(Chirp Spread Spectrum).*
Checkbox	Elemento de la interfaz gráfica de usuario que permite hacer selecciones múltiples de un conjunto de opciones.
Checksum	Suma de verificación. Es una forma de control de redundancia, una medida muy simple para proteger la integridad de datos.

DCE	Equipo de Comunicación de Datos (*Data Circuit-terminating Equipment*). Se considera DCE a todo dispositivo que participa en la comunicación entre dos equipos pero que no es receptor final ni emisor original de los datos que forman parte de esa comunicación.
Debugger	Depurador. Herramienta típica de un entorno de desarrollo que permite depurar el funcionamiento de un *firmware* y localizar errores o situaciones no controladas de éste.
DSSS	Técnica de espectro ensanchado por secuencia directa (*Direct Sequence Spread Spectrum*).
DSN	Número de secuencia de trama de datos (*Data Sequence Number*).
DTE	Equipo Terminal de Datos (*Data Terminal Equipment*). Es aquel componente del circuito de datos que hace de fuente o destino de la información.
EBCDIC	Código estándar de 8 bits propiedad de IBM (*Extended Binary Coded Decimal Interchange Code*).
ED	*Energy Detection.*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory.*
Endpoint	Punto final de la aplicación. Por lo general representa uno de los posibles puntos de acceso a una aplicación desde la subcapa de soporte de aplicación.
FCS	Secuencia de verificación de trama incorrecta (*Frame Check Sequence*).
FFD	Dispositivo de funcionalidad completa (*Full-Function Device*).
Firmware	Bloque de instrucciones de un programa, grabado en una memoria de tipo no volátil, que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo.
Flag	Uno o más bits que se emplean para almacenar un valor binario o código que tiene asignado un significado.
Flash	Tipo de EEPROM que permite que múltiples posiciones de memoria sean escritas o borradas en una misma operación.
GPIO	Puerto de entrada/salida de propósito general (*General Purpose Input Output*).
GTS	Porciones temporales de una supertrama que se asignan a una determinada comunicación entre dispositivos (G*uaranteed Time Slot*).
GUI	Interfaz gráfica de usuario (*Graphical User Interface*).
HAN	Red de Área del Hogar (*Home Area Network*).
I^2C	Bus serie IIC (*Inter-Integrated Circuit*).
IDE	Entorno de desarrollo integrado (*Integrated Development Environment*).
Imagen	En el contexto de este proyecto, se denominará “imagen” a una copia exacta (bit a bit) del *firmware* de un dispositivo, codificado en un formato concreto.
IFS	Separación temporal necesaria entre tramas para otorgar al receptor de tiempo suficiente para su proceso (*InterFrame Spacing*).
KBI	Interrupciones similares a las generadas por teclados o pulsadores (*Keyboard Interrupt*).
Layout	Esquema de un circuito integrado.
LED	Diodo emisor de luz (*Light-Emitting Diode*).
LLC	Control de enlace lógico definido por el estándar IEEE 802.2 (*Logical Link Control*). Representa la parte superior del nivel de enlace del modelo OSI.
LoRIS	Proyecto para la Localización en Redes Inalámbricas en aplicaciones Socio-sanitarias.

LQI	Indicador de la calidad del enlace (*Link Quality Indicator*).
LR-WPAN	Red de área personal inalámbrica de baja tasa de transmisión (*Low Rate – Wireless Personal Area Network*).
LSB	Bit menos significativo (*Less Significant Bit*).
MAC	Capa de control de acceso al medio (*Medium Access Control layer*).
MCU	*Microcontroller Unit.*
Mesh	Red mallada.
NIB	*Network Information Base.*
NPDU	Unidad de datos del protocolo de red (*Network Protocol Data Unit*).
MSB	Bit más significativo (*Most Significant Bit*).
OEM	*Original Equipment Manufacturer.*
OSI	*Open Systems Interconnection.*
OTAP	*On The Air Programming.*
PAN	Red de área personal (*Personal Area Network*).
PCB	Tarjeta de circuito impreso (*Printed Circuit Board*).
PDU	Unidad de datos del protocolo (*Protocol Data Unit*).
PHY	Capa física (*Physical layer*).
PIB	*PAN Information Base.*
Polling	Operación de consulta constante.
Profile	Perfil. Definición de un conjunto de variables y comandos dentro de un campo de aplicación determinado, que permitirán a diferentes vendedores crear productos ZigBee interoperables.
PSSS	Técnica de espectro ensanchado por secuencia paralela (*Parallel Sequence Spread Spectrum*).
RAM	Memoria de acceso aleatorio (*Random Access Memory*).
Reset	Proceso de devolver a condiciones iniciales un sistema. Habitualmente este objetivo se alcanza apagando y volviendo a encender la alimentación.
RF	Radio frecuencia (*Radio Frequency*).
RF4CE	Protocolo inalámbrico desarrollado conjuntamente con la *ZigBee Alliance* diseñado especialmente para aplicaciones sencillas dispositivo a dispositivo que no requieren de las funcionalidades completas que ofrecen las redes malladas ZigBee.
RFD	Dispositivo de funcionalidad reducida (*Reduced-Function Device*).
ROM	Memoria de sólo lectura (*Read Only Memory*).
RSSI	Indicador de fuerza/potencia de señal recibida (*Received Signal Strength Indicator*).
SAP	Punto de acceso a un servicio (*Service Access Point*).
Single Chip	Solución inalámbrica integrada en un único chip. Habitualmente el módem radio y el microcontrolador.
Sink	Sumidero. Por lo general referido a un nodo de la red encargado de la recolección de medidas de otros dispositivos del sistema.

SiP	Conjunto de circuitos integrados suministrados todos juntos dentro del mismo módulo o encapsulado (*System in Package*).
SKKE	*Symmetric-key key establishment.*
SMAC	*Simple-MAC*. Protocolo propietario de *Freescale* muy sencillo para la comunicación entre dispositivos inalámbricos punto a punto o en redes en forma de estrella. Se basa en la capa física de 802.15.4 y ha sido diseñado para dispositivos con muy poca memoria y capacidad de proceso.
Sniffer	Dispositivo capaz de capturar las tramas emitidas en un canal compartido.
SoC	Integración de todos los componentes de un sistema electrónico dentro del mismo chip *(System on Chip).*
SPI	Bus de interfaz de periféricos serie (*Serial Peripheral Interface*).
SSCS	Subcapa de convergencia entre el servicio de datos del nivel MAC de 802.15.4 (MCPS) y el control de enlace lógico (LLC) (*Service-Specific Convergence Sublayer*).
SSP	Proveedor de servicios de seguridad (*Security Service Provider*).
Stack	Pila. Se emplea con doble acepción: como segmento de memoria que utiliza una estructura de datos tipo LIFO para almacenar información sobre las llamadas a subrutinas actualmente en ejecución en un programa (pila de memoria) y como la estructura de capas que conforma la implementación concreta de un protocolo inalámbrico según el modelo OSI (pila ZigBee).
START	Señal de inicio de una comunicación.
STOP	Señal de fin de una comunicación.
Switch	Conmutador o pulsador.
SynkroRF	Protocolo inalámbrico de *Freescale* basado en 802.15.4 pero que incluye una capa de red sencilla y propietaria.
Target	Conjunto concreto de opciones de compilación y enlazado definido para un proyecto. Es habitual que un proyecto de CodeWarrior disponga de distintos *targets* cada uno configurado para distintos dispositivos o funciones.
Timeout	Tiempo de margen que se asigna a un suceso para completar su labor de forma exitosa.
UART	Transmisor-Receptor Asíncrono Universal (*Universal Asynchronous Receiver-Transmitter*).
Unicast	Envío de información desde un único emisor a un único receptor.
UWB	*Ultra-wideband.*
Warning	Alarma del compilador del CodeWarrior para alertar al programador de un posible error en el código generado.
Wifi	Protocolo para el envío inalámbrico de datos también conocido como estándar IEEE 802.11b.
WPAN	Red de área personal inalámbrica (*Wireless Personal Area Network*).
WSN	Redes Inalámbricas de Sensores (*Wireless Sensor Networks*).
ZC	Coordinador ZigBee (*ZigBee Coordinator*).
ZDO	Objeto de dispositivo ZigBee (*ZigBee Device Object*).
ZDP	Perfil de dispositivo ZigBee (*ZigBee Device Profile*).
ZED	Dispositivo final ZigBee (*ZigBee End Device*).
ZigBee	Estándar específico para redes de sensores inalámbricas de bajo consumo.

Apéndice D
Bibliografía y Referencias

[1] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)”.IEEE Std 802.15.4-2003. http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf (Octubre 2003)

[2] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)”.IEEE Std 802.15.4-2006. http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf (Septiembre 2006)

[3] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)”.IEEE Std 802.15.4a-2007. http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf (Agosto 2007)

[4] “ZigBee Specification”. Document 053474r06, Ver 1.0. Diciembre 2004. ZigBee Alliance.

[5] ”ZigBee Specification”. Document 053474r13. Octubre 2006. ZigBee Alliance.

[6] ”ZigBee Specification”. Document 053474r17. Octubre 2007. ZigBee Alliance.

[7] Sitio Web Oficial de la Z-Wave Alliance. http://www.z-wavealliance.org/

[8] Mike Foley,”The New Wireless Frontier”. Junio 2007. Bluetooth SIG.

[9] Sitio Web Oficial de la ZigBee Alliance. http://www.zigbee.org/en/about/

[10] “ZigBee Home Automation Public Application Profile (ZigBee Profile 0x0104)”. Ref. 053520r25. Octubre 2007. ZigBee Alliance.

[11] “ZigBee Smart Energy Profile Specification (ZigBee Profile: 0x0109)”. Ref. 075356r14. Mayo 2008. ZigBee Alliance.

[12] “Latest ZigBee Specification and Golden Units Now Available“. (Nota de Prensa). Enero 2008. Zigbee Alliance. http://www.zigbee.org/imwp/idms/popups/pop_download.asp?contentID=12543

[13] “ZigBee Smart Energy: The Standard for Energy Efficiency Available Now“. (Nota de Prensa). Junio 2008. ZigBee Alliance. http://www.zigbee.org/imwp/idms/popups/pop_download.asp?contentID=13532

[14] Sitio Web de Freescale. Kits de Desarrollo ZigBee y 802.15.4 (chips 1321x). http://www.freescale.com/webapp/sps/site/prod_summary.jsp ?code=1321x_Dev_Kits

[15] Sitio Web Oficial de Ember. Kits de Desarrollo ZigBee Insight (EM250 & EM260 chips). http://www.ember.com/products_zigbee_development_tools_kits.html

[16] Sitio Web Oficial de Texas Instruments (T.I.). Kits de Desarrollo ZigBee. http://focus.ti.com/analog/docs/gencontent.tsp ?familyId=367&genContentId=24196

[17] "ZigBee Home Automation: The New Global Standard for Home Automation". (Nota de Prensa). Noviembre 2007. ZigBee Alliance. http://www.zigbee.org/imwp/idms/popups/pop_download.asp ?contentID=12128

[18] C.Enrique Ortiz, "Introduction to OTA Application Provisioning". Noviembre 2002. http://developers.sun.com/mobility/midp/articles/ota/

[19] Sitio Web Oficial de NLaza Soluciones S.L. http://www.nlaza.es

[20] Sitio Web Oficial de WinMerge. http://winmerge.org/

[21] Sitio Web Oficial de RealTerm. http://realterm.sourceforge.net/

[22] P. Baronti, P. Pillai, V. W. C. Chook, S. Chessa, A. Gotta, and Y. F. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and Zigbee standards". Computer Communications, vol. 30, no. 7, pp. 1655–1695. 2007.

[23] "Sweden boasts world's first ZigBee city". (Nota de Prensa). Ref. 075304r00ZB_MWG. Octubre 2007. Ember.

[24] "ZigBee Cluster Library Specification". Ref. 075123r02ZB. Mayo 2008. ZigBee Alliance.

[25] Sitio Web Oficial de Microchip. http://www.microchip.com/

[26] Sitio Web Oficial de Atmel. http://www.atmel.com/

[27] Sitio Web Oficial de Freescale Semiconductor. http://www.freescale.com/

[28] Sitio Web Oficial de Jennic. http://www.jennic.com/

[29] Sitio Web Oficial de Motorola. http://www.motorola.com/es

[30] P. Lajsner. "Developer's Serial Bootloader for M68HC08 and HCS08 MCUs". Ref AN2295. Agosto 2006. Freescale.

[31] “802.15.4/ZigBee Embedded Bootloader, Rev 1.1”. Ref. 802154EBRM. Febrero 2006. Freescale.

[32] “Manual de Instalación ND07“. Ref. NLAZA-MAN-002_1.0. Abril 2009. NLaza Soluciones.

[33] “MC13211/212/213/214 ZigBee™- Compliant Platform 2.4 GHz Low Power Transceiver for the IEEE® 802.15.4 Standard plus Microcontroller Reference Manual. Rev 1.4“. Ref. MC1321xRM. Agosto 2009. Freescale.

[34] “CodeWarrior™ Development Studio 8/16-Bit IDE User’s Guide“. Septiembre 2005. Freescale.

[35] “Freescale BeeStack™ Application Development Guide“. Ref. BSADG, rev 1.1. Enero 2008. Freescale.

[36] “BeeKit Wireless Connectivity Toolkit User’s Guide“. Ref. BKWCTKUG, rev 1.9. Septiembre 2009. Freescale.

[37] “Freescale BeeStack™ Software Reference Manual“. Ref.BSSRM, rev 1.0. Marzo 2007. Freescale.

[38] “Freescale Platform Reference Manual“. Ref. FSPRM, rev. 1.0. Enero 2008. Freescale.

[39] “I^2C-bus specification and user manual. Rev.03“. Ref UM10204. Junio 2007. NXP. http://www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf

[40] “Using the 87LPC76X microcontroller as an I^2C bus master“. Ref. AN464. Enero 2000. Philips Semiconductors.

[41] “AT24C512.Two Wire Serial EEPROM Datasheet. Rev. 1116O–SEEPR–1/07“. Enero 2007. Atmel.

[42] Sitio Web de los foros de Freescale. http://forums.freescale.com/freescale/

[43] “Gestión de memoria HS08 (II), Fichero Motorola S19“. Ref. NLAZA-IT-009. Noviembre 2006. Nlaza Soluciones.

[44] “Hexadecimal Object File Format Specification. Revision A, 1/6/88”. Junio 1988. Intel.

[45] “HC(S)08 Compiler Manual“. Noviembre 2005. Freescale.

[46] “HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual“. Abril 2006. Freescale.

[47] “HC(S)08/RS08 Debugger Manual“. Mayo 2006. Freescale.

[48] R. G. Carranza, “Diseño y validación de una aplicación de configuración remota de dispositivos sobre 802.15.4”. Proyecto Fin de Carrera. Ingeniería Superior de Telecomunicación. Enero 2009. Universidad Carlos III de Madrid.

[49] “EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange“. Agosto 1969. Electronics Industries Association.

[50] “HCS08 Flash Integration for ZigBee and 802.15.4 Applications“. Ref. AN2770, rev. 1.3, Marzo 2006. Freescale Semiconductor.

[51] S. Ledford, “Non-Volatile Memory Technology Overview”. Ref. AN1837. 2004. Freescale Semiconductor.

[52] A. Wheeler, “Commercial Applications of Wireless Sensor Networks Using ZigBee”. IEEE Communications Magazine. Abril 2007. Ember Corporation.

[53] G. L. López, “Diseño y desarrollo de una pasarela de interoperabilidad 802.15.4/802.3 para aplicaciones de sensores“. Proyecto Fin de Carrera. Ingeniería Superior de Telecomunicación. Enero 2009. Universidad Carlos III de Madrid.

[54] J. Furness. “¿Tiene Futuro ZigBee?”. Artículo de la Revista Española de Electrónica (REE). Ed. Marzo 2008, pp. 80-81. Farnell.

[55] C. Buratti, A. Conti, D. Dardari, R. Verdone, “An Overview on Wireless Sensor Networks Technology and Evolution”. Agosto 2009.